Universidad
Carlos III de Madrid

Departamento de Informática

Optimización del proceso de integración continua del simulador de robótica Gazebo mediante mejoras sobre su sistemas de pruebas automáticas

PROYECTO FIN DE CARRERA

Autor: José Luis Rivero Partida
Tutor: Alejandro Calderón Mateos

Leganés, Julio de 2017

Agradecimientos

A la comunidad de código libre y abierto. Por demostrar que el conocimiento no debería nunca ser objeto de restricciones, privaciones o mercantilismo. Por demostrar lo imbatible que es la cooperación y la aportación colectiva.

A mis compañeros del IRI. Con especial mención al Humanoid Lab. Por enseñarme lo que era la robótica, la pasión por transmitirla y lo maravilloso que puede ser una institución pública al servicio de su sociedad.

A aquella profesora de matemáticas que me dijo que nunca llegaría a nada. Aquí estamos, en el segundo proyecto final de carrera.

A mi hermano. A mi madre y a mi padre: por obligarme a dar aquellas clases de verano de matemáticas y físicas sin las cuales sería imposible tener ninguna licenciatura y poder disfrutar del trabajo que me proporciona tanto placer. Parece poco así escrito pero es una vida entera.

A mi padre, otra vez. Que menos que mencionar dos veces a quién ha estado cada año repitiendome que tenía que escribir este documento.

A Paula. Por el amor. Escrito así parece poco, pero es una vida entera por delante.

A la abuela. Allá donde esté se que le haría tanta ilusión como le hizo el primer proyecto final de carrera que escribí.

Índice

1. Índice de figuras y tablas

Figuras

Tablas

Capítulo 2

2. Introducción

El capítulo aborda una breve introducción al proyecto detallando el origen, los objetivos y la estructura del documento.

2.1 Origen

El desarrollo del proyecto se enmarca dentro de las aportaciones realizadas por el autor en el marco del proyecto de código abierto Gazebo que cuenta con el soporte de la Open Source Robotics Foundation (OSRF), la cual tiene como dedicación principal la producción, promoción y mantenimiento de código abierto dedicado al mundo de la robótica [1].

Gazebo [2] es un simulador 3D multipropósito tanto de entornos interiores como de exteriores enfocado principalmente al mundo de la robótica. Puede simular gran parte de los efectos físicos presentes en la naturaleza, dispone de un gran número de sensores simulados así como una colección de escenarios virtuales y robots comerciales listo para su empleo.

Dentro de su actividad relativa al desarrollo de software, la cual incluye al proyecto Gazebo, la OSRF practica el modelo de integración continua, formulado por Martin Fowler en el año 2000 [3], que consiste en la minimización del esfuerzo de integración en cada fase del desarrollo y la capacidad de poder utilizar el software en cualquier momento del mismo. Como parte fundamental de la implementación de este modelo de integración continua se encuentra la necesidad de disponer de diferentes mecanismos de pruebas automáticas que sean fácilmente ejecutables y reproducibles.

El origen del proyecto comienza en el análisis de la implementación del sistema de pruebas automáticas del que la OSRF disponía al comienzo de este trabajo. Este sistema inicial estaba basado en una implementación muy simple de los entornos de pruebas (chroot) y una configuración mínima de un servidor de integración continua, empleando el software Jenkins (ver figure 1).

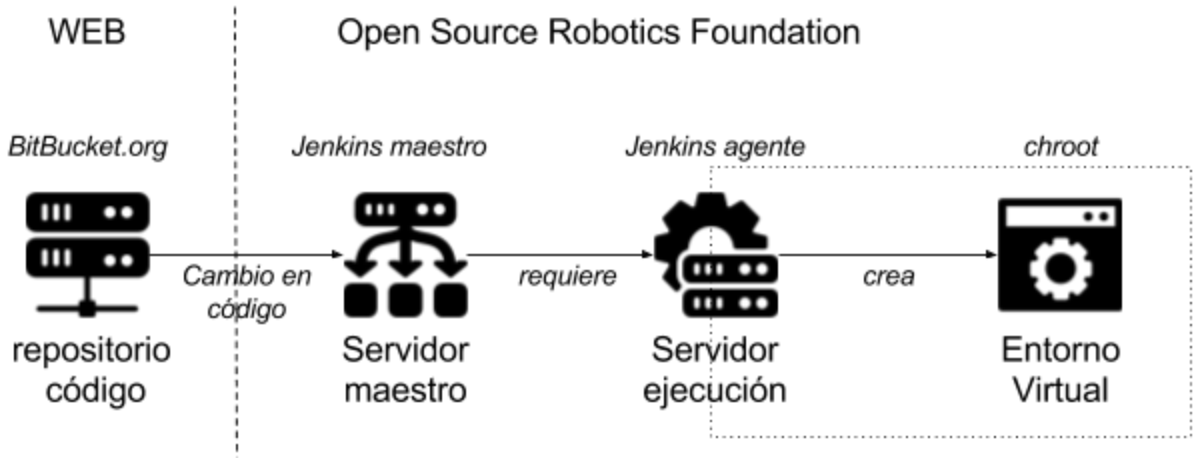


Figura 1: sistema de pruebas inicial del proyecto Gazebo

Este despliegue inicial de infraestructura destina a pruebas automáticas estaba lastrado por varios problemas serios, que hacían que el ciclo de integración continua fuera un freno al desarrollo ágil del proyecto: dificultades para el mantenimiento del código que ejecutaba las pruebas automáticas, ineficiencia en la creación de los entornos de pruebas, incapacidad de ejecución de pruebas automáticas que requieren de hardware específico o la falta de un diseño que permita escalar el número de plataformas y versiones del simulador soportadas por el ciclo de integración continua.

Este documento describe alguna de las soluciones que el autor aportó al proyecto para poder superar la problemática descrita y contribuir con el que es probablemente el simulador más popular en la comunidad robótica de código abierto.

2.2 Objetivos

El principal objetivo del proyecto es la mejora, en diferentes aspectos, de la implementación técnica y los procesos que afectan al ciclo de integración continua y pruebas automáticas existentes inicialmente, tras llevar a cabo el análisis de la situación inicial y de las posibles opciones de mejora existentes.

Es importante indicar que está fuera del alcance de este proyecto el replantear el diseño y los pasos del propio proceso de integración continua en sí, simplemente se aborda el proceso de optimización desde el punto de vista puramente técnico de la implementación del diseño existente.

Aspectos que este proyecto pretende mejorar:

- Mejorar el desarrollo y mantenimiento del código empleado en el servidor de integración continua para dotarlo de características deseables de las que actualmente carece: histórico de cambios, reproducibilidad, escalabilidad, etc.

- Aproximar las condiciones del entorno de ejecución de integración continua a los sistemas en producción (sistemas finales de empleo del simulador) mediante la mejora de los entornos de pruebas, su aislamiento e interacción con el hardware.

- Reducir los tiempos empleados por los ciclos de compilación mediante la reutilización de los entornos de pruebas generados siempre que sea posible.

- Evaluar si existe alguna alternativa mejor al software que implementa el servidor de integración continua existente en el inicio del proyecto. En caso de que sea la mejor opción, buscar cómo explotar funcionalidades nativas del mismo para la consecución del resto de objetivos descritos en este proyecto.

En resumen: llevar a cabo el análisis de la situación inicial respecto a diferentes aspectos del ciclo de integración continua, focalizando sobre su implementación técnica, y razonar qué opciones existen como alternativas para mejorarlo, proponiendo la mejor en cada caso y ejecutando su implementación.

2.3 Estructura del documento

Previo a la exposición detallada punto por punto de los capítulos contenidos en este documento, una pequeña mención a una particularidad de este proyecto: el trabajo incluye cuatro subproyectos (mejora no interrelacionadas) que están estructurados como si se tratara de proyectos independientes, con sus propias secciones que contextualizan la mejora, describen el análisis, aportan la solución y desarrollan las conclusiones. El autor cree que resulta más cómodo para el lector disponer de toda la información compacta (en espacio) y no tener que recorrer largas secciones. No obstante, se incluyen, secciones generales que abordan temas transversales a estos subproyectos.

El presente documento se estructura en los siguientes capítulos y secciones:

- **Introducción:** detalle del origen y los objetivos del proyecto. Incluye este capítulo que documenta la propia estructura del documento completo.
- **Estado de la cuestión:** se presenta el software principal, el simulador Gazebo, sobre el que trata en gran parte el proyecto así como las características más relevantes que afectan al objetivo principal del proyecto. Además, se describen las posibles soluciones técnicas que a distintos aspectos transversales a todo el proyecto:

- **Subproyectos:** el proyecto presenta cuatro subproyectos en los cuales se analizan y ejecutan distintas mejoras sobre el estado inicial de la implementación del proceso de integración continua. Cada capítulo de subproyecto tiene las siguientes secciones:
    - Descripción: introducción al subproyecto que se aborda, el contexto dentro del proyecto general y su relación con el simulador Gazebo.
    - Objetivo: el objetivo principal del subproyecto.
    - Estado inicial: descripción técnica del estado inicial del que parte el subproyecto y se pretende mejorar.
    - Soluciones existentes: análisis de soluciones posibles al problema planteado, haciendo especial énfasis en la parte técnica de las mismas. Puntos fuertes y puntos débiles de cada una.
    - Solución propuesta: razonamiento de la elección de la solución final de entre las propuestas y detalles sobre su implementación.
    - Conclusión: reflexión sobre la mejora efectuada y justificación sobre la elección de la solución ejecutada así como su impacto en el sistema de integración continua.
- **Presupuesto:** análisis económico de los costes de ejecución que implicaría la puesta en marcha del proyecto detallado en este documento.
    - Coste del personal para desarrollo e implementación: consideraciones sobre los costes que afectan a la contración de personal en base a una estimación de tiempos sobre las distintas tareas.
    - Sistemas informáticos: coste estimado de la instalación de un sistema de integración continua válido para este proyecto.
    - Coste total: cálculo del coste total del proyecto incluyendo amortizaciones y costes indirectos.
    - Impacto socioeconómico: breve análisis sobre los diferentes efectos sociales, económicos y ambientales que derivan del proyecto.
- **Conclusiones generales:** reflexiones transversales sobre el trabajo expuesto en el documento en su totalidad.
- **Futuras líneas:** explicación razonada de posible áreas de mejora a realizar que han quedado fuera del alcance de este proyecto.
- **Marco regulador:** corta exploración de qué aspectos legislativos afectarían al proyecto.
- **Bibliografía:** referencias documentales utilizadas al realizar este documento.
- **Anexos:** añadidos de documentación, código fuente, esquemas, etc. que sirven para complementar lo expuesto en cada capítulo.

El código fuente generado durante el desarrollo de este proyecto se encuentra disponible de manera pública en: https://github.com/j-rivero/pfc-computer-engineering

Capítulo 3

3. Estado de la cuestión

Este capítulo presenta en tres grandes bloques los aspectos del proyecto que afectan a varios de los subproyectos que se encuentran en los capítulos cuatro, cinco, seis y siete. Cada subproyecto está dotado de su propia exposición sobre el estado de la cuestión repartida entre los puntos de estado inicial y soluciones existentes.

3.1 Gazebo: el simulador de robótica objeto de testing

El proyecto aborda el estudio y la mejora del proceso de integración continua y ejecución de pruebas automáticas de un software determinado: Gazebo. Aunque los procesos de pruebas comparten un buen número de aspectos, hay diferencias importantes dependiendo de la naturaleza del software objeto de pruebas. En este capítulo se describe brevemente la historia y el desarrollo de Gazebo y se detallan las cuestiones técnicas que afectan al diseño e implementación del entorno de pruebas así como a la ejecución de las mismas.

3.1.1 El proyecto Gazebo

El desarrollo de Gazebo comienza en el año 2002 y está basado en la creación de un simulador 3D con un alto grado de fidelidad ante la necesidad de simular robots en entornos abiertos de naturaleza variada. Desde fases tempranas muchos usuarios lo utilizaron con éxito también para entornos interiores.

Gazebo está considerado código abierto, su código emplea la licencia Apache2. Inicialmente se desarrolló como resultado de la tesis doctoral de Nate Koening en la Universidad de Southern California [4]. El autor original fue posteriormente contratado por una de las empresas clave en el desarrollo de la robótica open source, Willow Garage y cuando ésta desapareció pasó a estar soportado dentro de la Open Source Robotics Foundation.

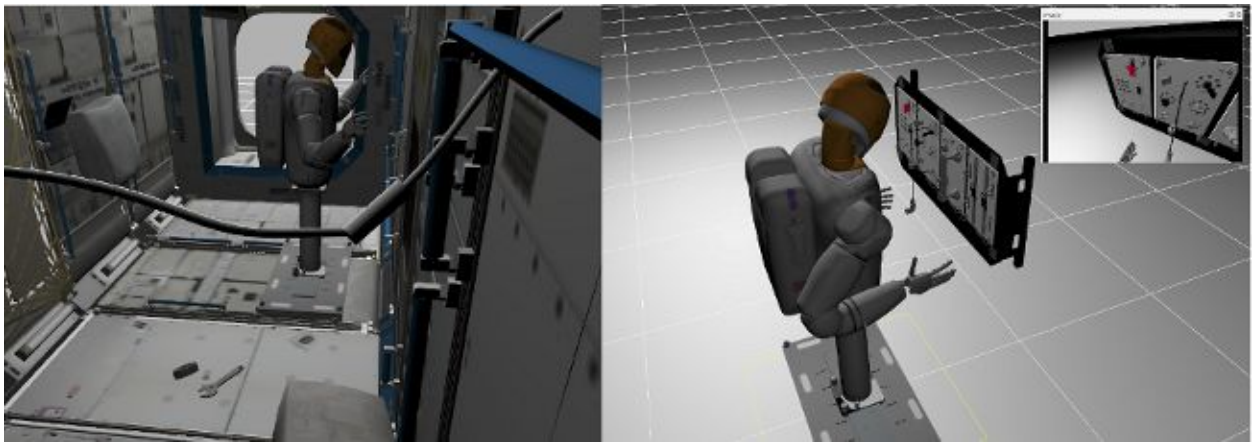


Figura 2: renderización de la simulación del robot de la NASA Robonaut 2 en Gazebo

Técnicamente, Gazebo fue diseñado para poder recrear entornos 3D dinámicos con uno o varios robots interactuando al mismo tiempo. Es capaz de utilizar distintas librerías de cálculo de física, proporciona una gran variedad de sensores virtuales listos para ser utilizados (cámaras, láseres, sensores de contacto, infrarrojos, etc.), dispone de una interfaz de usuario que permite interactuar con la simulación mientras se ejecuta y también proporciona herramientas gráficas para la creación de modelos 3D sencillos.

3.1.2 Aspectos técnicos que condicionan el testing

Un análisis de Gazebo posicionándolo fuera del contexto puro de capacidades robóticas y visto como cualquier otro producto de software genérico, arroja algunos detalles que son determinantes a la hora de influir en todo el proceso de integración continua.

Gazebo es un proyecto escrito **fundamentalmente escrito en C++** que ha aumentado el total de líneas de código que lo componen, desde las ciento ochenta y seis mil en la versión 1.9 a las casi trescientas mil en su última versión 8. Dispone de diferentes niveles de pruebas automáticas, desde tests unitarios que cubren buena parte de sus clases (alrededor del 50% de todas las funciones) hasta test de integración o regresión. En total cuenta en la actualidad con **más de mil doscientos tests** sumando las diferentes categorías.

Figura 3: ventana principal de Gazebo y subventana con el renderizado generado por la cámara acoplada al dron

El simulador Gazebo depende de diferentes librerías que sirven de apoyo para implementar todas sus funcionalidades. En la última versión podemos encontrar alrededor de **cuarenta dependencias distintas [5]**, entre las obligatorias y los soportes opcionales en tiempo de configuración. Desde software básico para este tipo de aplicación y el lenguaje de programación en que está escrito (pkg-config, Boost, Tbb, Doxygen, Qt5, libtar, Ogre) hasta algunas bibliotecas relacionadas íntimamente con el área de la robótica (SDFormat, Bullet, ODE, Gdal, Simbody, etc.)

La arquitectura de la aplicación [5] está basada en un modelo **cliente <-> servidor [6]** que se comunica a través del sistema de red.

- El servidor de Gazebo, llamado *gzserver*, ejecuta la simulación de la física y la generación de los datos de los sensores simulados, como pueden ser cámaras o infrarojos. El servidor está considerado como la parte clave del simulador y puedes ser ejecutado sin un interfaz gráfico de usuario.

- El cliente de Gazebo, llamado *gzclient*, es un interfaz gráfico (emplea la librería QT) de usuario que permite la visualización e interacción con varios aspectos de la simulación que ejecuta el servidor. Puede existir más de un cliente por servidor.

El servidor realiza las tareas de simulación de sensores. En algunos casos, como aquellos sensores íntimamente relacionados con la imágen o la luz (cámaras o los láseres), *gzserver* va a necesitar de un **servidor gráfico** (por ejemplo, sistema X en Linux) para renderizar la simulación de estos sensores. Esto hace que la existencia de un servidor X sea no sólo necesaria para realizar procesos de pruebas sobre el cliente sino también sobre el propio servidor, si queremos desplegar el abanico de pruebas sobre todas las funcionalidades del simulador.

Gazebo puede funcionar en entornos gráficos sin necesidad de tarjetas gráficas potentes o grandes unidades de procesamiento gráfico (GPUs) aunque su rendimiento puede beneficiarse en gran medida de la presencia de las mismas. Históricamente, la plataforma de referencia para las pruebas automáticas y el desarrollo ha utilizado tarjetas NVIDIA y parte del código que interactúa con el **subsistema OpenGL** ha estado probado fundamentalmente con esta marca comercial de tarjetas gráficas a través de los drivers propietarios que la compañía proporciona en los entornos derivados de Debian, como puede ser Ubuntu.

Como excepción, Gazebo dispone de tipos específicos de sensores diseñados especialmente para aprovechar toda la potencia de una **GPU** y solamente podrán funcionar bajo la presencia de la misma. En el contexto de este proyecto, si se quiere completar el total de las pruebas automáticas, la presencia de una GPU en el sistema de testing es fundamental para un completo ciclo de integración continua.

3.1.2.1 Software complementario a Gazebo

Dentro de los aspectos técnicos que condicionan las pruebas automáticas de Gazebo están las propias dependencias que tiene el simulador. Dentro de estas dependencias, como se ha descrito en la sección previa, se pueden categorizar en diversos grupos con llamativo protagonismo de dos principales: bibliotecas auxiliares frecuentes en los proyectos C++ y bibliotecas relativas al entorno de la robótica .

En este segundo grupo de software relacionado con el ámbito de la robótica cabe destacar que algunas de las bibliotecas y proyectos nacieron como parte de los procesos de refactorización de Gazebo, que inicialmente estaba compuesto monolíticamente por más de ciento ochenta y cinco mil líneas de código. Desde la versión 1.8, el simulador ya se utilizaba una de estas bibliotecas, conocida como SDFormat (Simulation Description Format) [7]. Ésta y otras librerías nacieron como proceso de generar proyectos más pequeños que sean reutilizables y compartidos con el resto de la comunidad de robótica open source.

El proyecto, y librería del mismo nombre, SDFormat estaba y está desarrollado por el mismo equipo de desarrolladores que Gazebo e implícitamente soportado por bajo la estructura de la Open Source Robotics Foundation.

Desde del ciclo de desarrollo de la versión 6 de Gazebo, finales de 2015, la apuesta por externalizar partes del código de Gazebo se intensificó y se crearon diferentes librerías bajo el proyecto denominado *"ignition robotics"*. La primera de ellas contiene el código encargado de implementar funciones y operaciones matemáticas (*ignition-math*). A de esta primera biblioteca le siguieron otras dos: la capa de transporte que utilizaba el simulador para comunicar al servidor con el cliente (*ignition-transport*) y el código que implementa el conjunto de mensajes que se intercambian en la capa de transporte (*ignition-msgs*).

3.2 Servidores de integración continua

Un servidor de integración continua es un paquete software que ayuda a ejecutar los procesos de integración continua diseñados. Es una pieza clave en la implementación de la estrategia de testing formando parte de uno de los principìos fundamentales de los procesos CI: la automatización de la generación del producto final software para cada uno de los cambios que se realizan en su código base, así como la ejecución de mecanismos de pruebas automáticos proporcionados por el propio software. También entran dentro de las funciones principales de un servidor de integración continua las tareas de informe de errores o cambios a los propios desarrolladores así como al equipo de testing.

Los procesos de integración continua (ver figura 4) están basados en varios pasos que se podrían resumir en:

- **Obtención del código fuente:** el código fuente suele proceder de uno o varios sistemas de control de versiones (git, mercurial, subversion, etc.).
- **Compilación o generación de código:** depende del tipo de software al que se someta a los procesos de integración continua será necesario un procesamiento para obtener los artefactos finales del mismo, que pueden ser ejecutables, páginas web, bibliotecas u otro código auxiliar.
- **Ejecución de pruebas automáticas:** cada producto software debería incluir, como principio de buenas prácticas, un conjunto de pruebas automáticas con el objeto de verificar si efectivamente el software obtiene el resultado esperado por el usuario. El servidor de integración continua debe ser capaz de crear un entorno donde estas pruebas automáticas puedan ejecutarse, así como llevar a cabo la ejecución de las mismas.
- **Generación de informes sobre resultados:** como comunicación que debe llegar al equipo de desarrollo y/o al de testing, el servidor debe ser capaz de generar una informe lo suficientemente explicativo para que las diferentes personas involucradas en el desarrollo del mismo. Puede incluir artefactos generados durante la compilación.

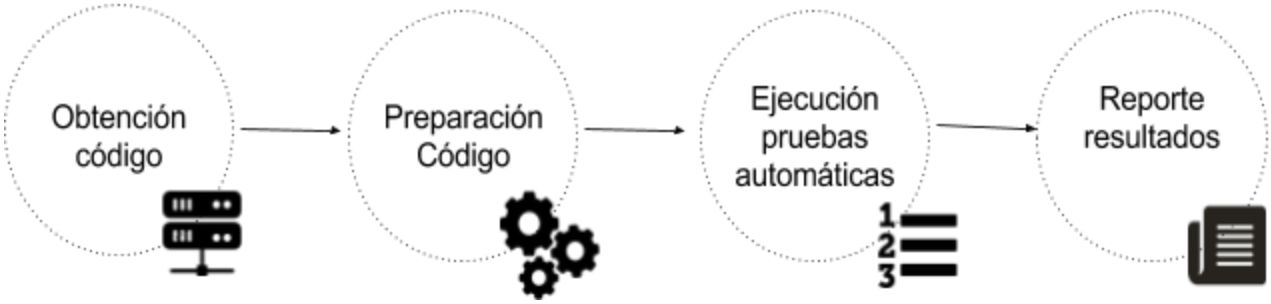


Figura 4: pasos de un proceso de integración continua

3.2.1 Hudson/Jenkins

Hudson es un servidor de automatización de código abierto (inicialmente con licencia MIT, hoy en día Eclipse 1.0), programado en Java que ofrece a los equipos de desarrollo la integración de diferentes sistemas de gestión de código con diferentes métodos de compilación o generación de código. El versión inicial se liberó en el año 2005 siendo Kohsuke Kawaguchi el principal desarrollador contratado por Sun Microsystems.

En 2010 Oracle compra Sun Microsystems y se produce una disputa por quién controla la marca y el nombre “Hudson”. Los desarrolladores principales, con Kohsuke a la cabeza, deciden renombrar el proyecto, creando un fork, al que llamaron Jenkins.

Dado que ambos proyectos comparten gran parte del código original, multitud de funcionalidades y diseño de interfaz, este proyecto se focalizará únicamente en analizar el que ha tenido y tiene un mayor desarrollo durante los últimos años: **Jenkins.**

Al compartir la base de código del proyecto original (Hudson), Jenkins es un servidor de integración continua también programado en Java y open source, heredó la licencia MIT que tenía su código original. En 2014 el creador de Kohsuke Kawaguchi fue contratado como CTO (Chief Technical Officer) por la empresa CloudBees, dedicada a la provisión de servicios Java en la *nube*. Desde entonces, CloudBees ha sido la principal empresa de provisión de servicios Jenkins comerciales y también que ha financiado gran parte del desarrollo del código de Jenkins.

De las características de las que dispone Jenkins como servidor de automatización destacan:

- Fácil instalación: un único fichero Java (.war) que se aloja en contenedor Servlet. Existen paquetes e instaladores en multitud de plataformas.
- Integración con amplia variedad de sistemas de control de versiones: CVS, SVN, Git, Mercurial, etc.
- Muy flexible en la creación de todo tipo de tareas automatizadas.
- Generación de informes de errores a partir de los resultados de los test automáticos.
- Notificar por diferentes medios a las distintas partes interesadas del resultado de la integración.
- Ejecutar el despliegue del código generado tanto en servidores en producción como en entornos de test.
- Coordinación de sistema de diferentes plataformas: funciona, tanta para su nodo maestro como el resto de los agentes, en Linux, BSD, MacOSX, Windows, etc.

3.2.1.1 Glosario de términos

- **Agente:** un sistema informático (una máquina real o virtual) que se conecta al Nodo maestro y ejecuta las tareas que éste le envía.
- **Ejecución:** hace referencia al resultado de una ejecución concreta de un proyecto Jenkins
- **Extensión o plugin:** un pequeño proyecto software integrado en Jenkins que le proporciona características o funcionalidades adicionales y que puede estar desarrollado y mantenido por un usuario que no forma parte del equipo de desarrollo principal.
- **Nodo:** un sistema informático (una máquina real o virtual) que forma parte de Jenkins y es capaz de ejecutar proyectos. Tanto el sistema maestro (master) de Jenkins como los agentes son considerados nodos.
- **Maestro o Master**: el nodo y proceso central de Jenkins que almacena las configuraciones, carga los plugins y muestra el interfaz web de Jenkins.
- **Proyecto:** una configuración proporcionada por el usuario que describe un proceso concreto que Jenkins debe ejecutar, como pueda ser la compilación de una pieza de software, el empaquetado de un software ya existente o creado por otro proyecto, o la realización de alguna tarea periódica sobre alguno de los sistemas conectados a Jenkins.
- **Trabajo:** sinónimo de Proyecto.

3.2.1.2 Cómo funciona Jenkins

Un sistema Jenkins está compuesto de al menos un nodo maestro (master) y, opcionalmente, un número indeterminado de agentes que están bajo la coordinación de los nodos maestros. Tanto el nodo maestro como el resto de agentes pueden ejecutar diferentes procesos definidos previamente por el usuario conocidos como proyectos.

El **nodo maestro** es la pieza central de todo sistema Jenkins. En la mayoría de casos existe un único nodo maestro pero en instalaciones grandes o las que requieren de una alta disponibilidad pueden existir un gran número de ellos. El nodo maestro almacena todos los parámetros de configuración general sobre el sistema Jenkins, usuarios y credenciales de las partes que interactúan con el servidor, gestiona y controla todas las extensiones o plugins instalados, dispone de los registros (logs) de todos las actividades de la instalación, etc.

Cada proyecto en Jenkins dispone de una configuración propia, que puede incluir: los permisos de autorización para usuarios del servidor, si se pueden lanzar ejecuciones paralelas del propio proyecto, si únicamente se puede ejecutar un tipo concreto de agentes, etc. Además de la configuración, pueden incluir una gran variedad de acciones: obtener código fuente de un repositorio, ejecutar comandos shell, copiar artefactos resultantes del proceso (paquetes binarios, resultados de tests, etc.) a otros procesos u otros sitios web, enviar notificaciones en diversos tipos de canales (correo electrónico, IRC, etc.) a los usuarios involucrados en el proyecto, etc.

Los **agentes** en Jenkins son sistemas (Linux, Windows, Mac, …) conectados al maestro que se encargan de ejecutar los proyectos configurados por el usuario en función de los eventos definidos en la propia configuración.

Al ser un software dedicado a un gran número de tareas diferentes que pueden configurarse gracias a la flexibilidad que ofrece cada proyecto que se crea y la gran cantidad de plugins que pueden proporcionar funcionalidades extra, no es trivial escoger un sólo caso de uso para explicar cómo funciona, en una determinada ejecución, una instancia de Jenkins ya que puede ser muy variable.

Un **caso de uso típico**, y el que originalmente estaba en vigor en el proyecto Gazebo cuando el proceso de mejora que describe este documento comenzó, es el integrar el servidor Jenkins en un ciclo de integración continua lanzando cada ejecución del mismo cuando un usuario envía una petición de inclusión de código (pull request) a un repositorio de código alojado en alguna plataforma web (github, bitbucket, etc.).

La secuencia de acciones e interacciones entre el repositorio de código web y el servidor Jenkins es sencilla (ver figura 5):

I. La **petición de inclusión de código** (pull request) llega a la plataforma que aloja el repositorio de código.
II. La plataforma está configurada para **enviar la información** (metadatos generalmente en formato JSON) sobre la petición **al servidor Jenkins**, que la recibe.
III. Cada proyecto definido en Jenkins proporciona información sobre qué tipo de peticiones le afectan (repositorio de origen, ramas en el mismo, etc.). **Cada proyecto afectado por la nueva información comenzará una nueva ejecución** que suele incluir pasos como: obtención del código, compilación y ejecución de pruebas automáticas.
IV. La ejecución del proyecto de **Jenkins envía el resultado** de la misma al sitio web responsable del repositorio de código.
V. La plataforma web **actualiza el estado de la integración continua** en la petición de inclusión de código.

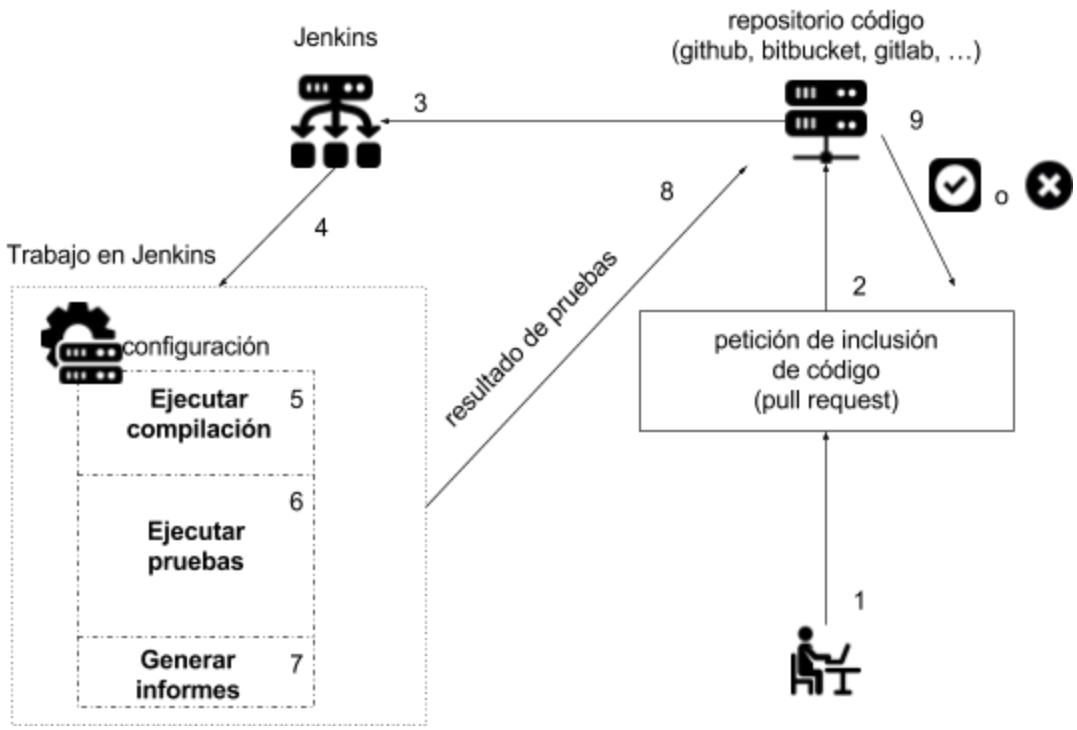


Figura 5: diagram detallado de interacción de un proceso de integración continua utilizando Jenkins

3.2.1.3 Flexibilidad y personalización en Jenkins

Uno de los puntos fuertes de Jenkins como proyecto es la capacidad de personalización de que dispone. El sistema está diseñado para ser totalmente personalizable en muchos de los pasos y procesos internos. Para esto, existe un sistema de extensiones (plugins) [8] pensado para extender y ampliar las funcionalidades de Jenkins a través de múltiples herramientas: diferentes herramientas de generación de código, interacción con proveedores de servicios en la nube, herramientas de análisis, entre otras.

La mayor parte de estas extensiones son liberadas por los usuarios y están perfectamente integradas en Jenkins a través del centro de extensiones (Update Center) que se encarga no sólo proporciona un método de instalación sencillo sino que permite descargar las actualizaciones de los plugins cuando están disponibles.

La cantidad de plugins compartidos creados por parte de usuarios, administradores o desarrolladores que están disponibles añaden un abanico de posibilidades en múltiples campos del servidor que hacen que en la mayoría de los casos solucionar una carencia sea tan sencillo como buscar el plugin existente que ya lo hace.

3.2.2 BuildBot

BuildBot es un software diseñado para la automatización de sistemas complejos, incluyendo aquellos encargos de la integración continua. BuildBot está programado en python y emplea principalmente el framework Twisted, es código abierto y está licenciado bajo GPL2. Es multiplataforma pudiendo ejecutarse en Linux, Windows, MacOSX, etc.

La primera versión del proyecto data de 2003 y en Junio de 2017 continuaba como proyecto activo. Es empleado por proyectos como Mozilla, Python o Chromium entre otros. Algunas de sus principales características:

- Fácil instalación: empleando el instalador de python pip se puede fácilmente instalar en pocos minutos.
- Ficheros de configuración programables: la configuración de cada trabajo está codificada en python, pudiendo emplearse todas las funcionalidades del mismo para generar configuración.
- Múltiples mecanismos de notificación: correo electrónico, interfaz web, IRC, etc.
- Integración con sistema de control de versiones: SVN, GIT; Mercurial, etc.
- Funcionalidades extra: disponibles mediante el empleo de plugins

Un sistema BuildBot (ver figura 6) dispone de uno o más sistemas maestros (*masters*) y una colección de sistemas ejecutores (*workers*). El sistema funciona como si se tratara de planificador de tareas: encola los trabajos pendientes y los ejecuta cuando tiene los recursos disponibles. Una vez hecho esto, informa sobre el estado de la ejecución. Toda la coordinación se realiza en los Masters y las acciones de ejecución en los sistemas workers.

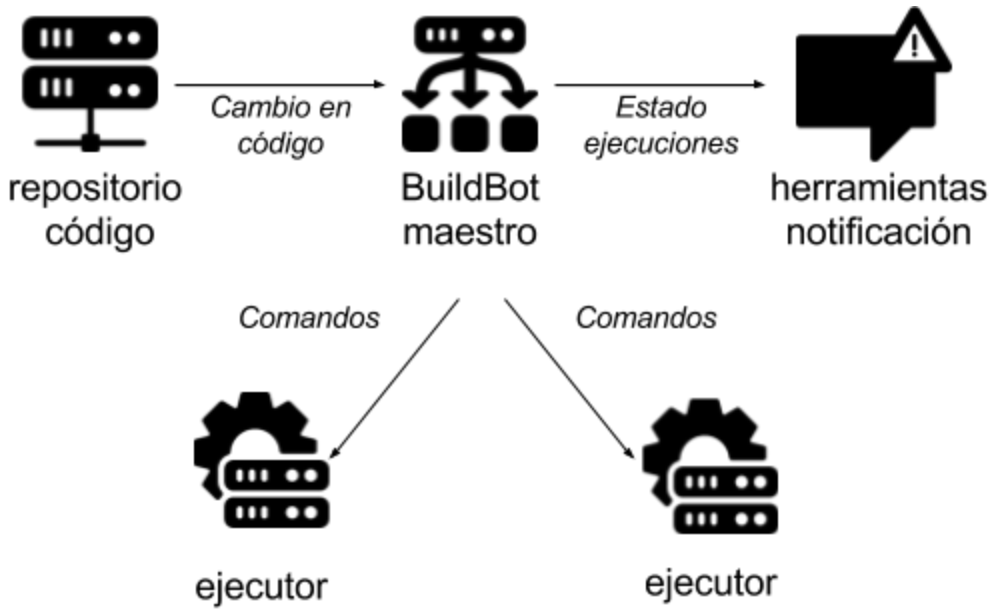


Figura 6: diagrama de componentes empleados por el servidor de automatización BuildBot

La filosofía y diseño del sistema es, en su esencia (nodo principal, nodos ejecutores, lista de tareas) **muy similar a la que se ha detallado para Jenkins en el apartado anterior**. De hecho, el diagrama de interacción con un repositorio de código aplicaría sin cambios relevantes para un sistema BuildBot.

3.2.3 Servicios en la nube para automatización: Travis.io

Travis es un servicio de integración continua dedicado a la integración de proyectos hospedados en el servidor de código github. Soporta un gran número de lenguajes de programación y otras herramientas de configuración [9], entre los que están múltiples versiones de PHP, Node.js, npm, etc. Es gratuito para los repositorios públicos de github y existe una versión de pago para los repositorios privados. Es código abierto (licencia MIT). El proyecto está soportado por una compañía alemana creada en 2011.

A diferencia de Jenkins y BuildBot, Travis ofrece el conocido “Software como servicio” (Software as a Service - Saas) como una solución integral a la infraestructura y configuración de testing. El servicio de Travis está íntimamente ligado al servicio de alojamiento de código github.

3.2.3.1 Cómo funciona Travis

El funcionamiento de Travis es completamente distinto al que previamente se ha detallado para Jenkins o BuilBot. Al ser un servicio web requiere al usuario registrarse para poder usarlo y dotar a Travis de los permisos necesarios para acceder a los repositorios GitHub a los cuales se quiere dotar de integración con este servicio de pruebas automáticas.

La segunda parte es la configuración de las acciones a realizar en el ciclo de pruebas automáticas (configuración, compilación, ejecución de pruebas, etc.) y la propia configuración del entorno virtual de pruebas (Travis utiliza internamente Docker para los sistemas de pruebas sobre Linux). Esto se realiza por cada uno de los distintos repositorios mediante la inclusión en el propio código fuente de un fichero llamado *.travis.yml*.

El fichero *.travis.yml* es la pieza fundamental de interacción con los servidores de Travis. Es un fichero en formato YAML (especializado en la serialización de datos que puedan ser leídos fácilmente por el ser humano) que contiene las instrucciones y configuración del entorno de pruebas. Entre las configuraciones que se pueden definir existen las referentes al tipo de lenguaje de programación empleado por el software, el tipo de compilador, la versión de Ubuntu a utilizar, etc. Las acciones se dividen en secciones que se ejecutan en distintos momentos del proceso de creación del entorno de pruebas y de la ejecución de las mismas.

Un ejemplo de fichero*.travis.yml* para un hipotético software c++ que emplea cmake podría ser:

```
sudo: required
before_install:
  - sudo apt-get install -y sdformat libboost-dev libogre-dev libtinyxml-dev
  - sudo apt-get install -y libcurl4-openssl-dev

# Enable C++ support
language: cpp

# Compiler selection
compiler:
  - gcc

# Build steps
script:
  - mkdir build
  - cd build
  - cmake ..
  - make -j
```

Con este fichero en el repositorio, en cada cambio registrado en el sistema de control de versiones (commit) que alcance el servidor Travis lanzará una ejecución de pruebas con los pasos y configuración contenidos en el fichero .travis.yml.

3.2.3.2 Otras alternativas SaaS

Durante el año 2015 y sobre todo 2016 múltiples opciones similares a Travis sufrieron un gran crecimiento y popularidad.

Todas estas plataformas Saas funcionan de manera muy similar mediante el empleo de un fichero de configuración incrustado en el propio código. Inicialmente se diferenciaron en los entornos de pruebas soportados (por ejemplo AppVeyor trabaja únicamente con Windows), en las plataformas en las que se integran (Drone.io funciona tanto en Bitbucket como en Github) en la oferta de planes tanto gratuitos como comerciales, en los interfaces que muestran la información y en otros aspectos no relacionados intrínsecamente con el diseño o la manera de operar. Con el paso del tiempo muchas de ellas soportan ya varias plataformas.

Algunas de las opciones que operan de igual manera que Travis son: CircleCI, Drone.io, AppVeyor, Snap CI, Semaphore CI, etc. **Existen otras muchas focalizadas en otro tipo de interacciones con el código** como puedan ser aquellas dedicadas al análisis de qué parte del código está tratada en las pruebas automáticas del mismo (Coveralls, Coverity, etc.) o a analizar la calidad del código (Landscape, Codacy, etc.). No está entre los objetivos de esta sección detallar cada una de ellas

3.3 Entornos virtuales para ejecución de pruebas

Un entorno de ejecución de pruebas automáticas consiste en una configuración software y/o hardware en el cual un equipo de desarrollo prueba una nueva versión de su software. Generalmente consiste en una versión simplificada de un entorno de producción que ayuda a encontrar errores en fases de pre-produción [10]. Para implementar con éxito algunos de estos factores, los procesos de integración continua utilizan en muchas ocasiones técnicas de virtualización. Se detallan a continuación.

3.3.1 Virtualización

Virtualización es un término genérico que engloba el hecho de replicar como si fueran reales diferentes ámbitos relativos a la arquitectura de computadores y los sistemas operativos. En el presente documento la virtualización hace referencia a la **virtualización hardware**. La virtualización hardware tiene por misión la (re)creación de una máquina virtual que pueda albergar un sistema operativo igual que lo haría una computadora real.

3.3.2 Paravirtualización

Dentro de la virtualización hardware encontramos una de sus evoluciones conocidas como **paravirtualización**: una técnica introducida por el proyecto Xen en la cual el sistema operativo virtualizado conoce la existencia de una instancia intermedia (el *hipervisor*) e interactúa con los interfaces proporcionados por ésta en lugar de emitir instrucciones hardware directamente. Esto proporciona importantes ventajas en los tiempos de ejecución de la simulación. Dentro de la paravirtualización existen dos subcategorías: la **paravirtualización nativa**: en la que el hipervisor actúa también como sistema operativo anfitrión y la **paravirtualización albergada**, en la que el software de virtualización se instala en un sistema operativo base, como cualquier otro software:

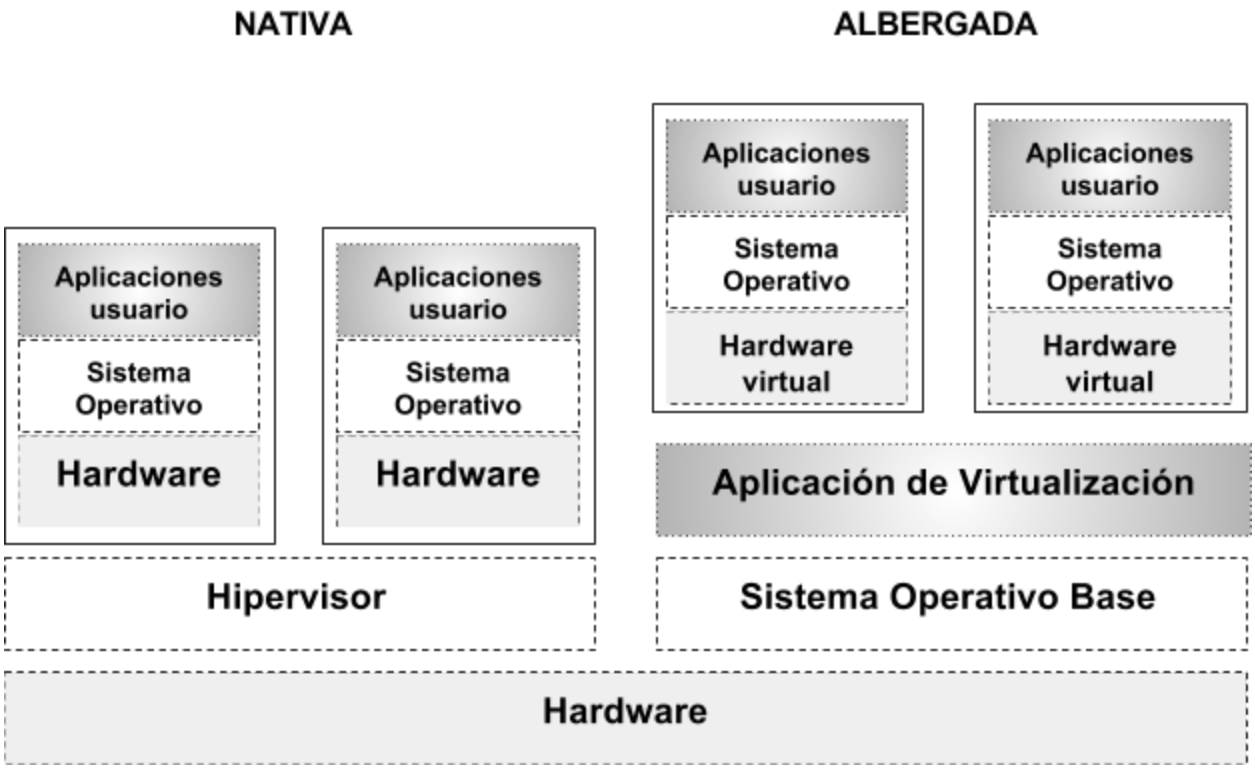


Figura 7: comparación de capas lógicas de sistemas paravirtualizados nativos y albergados

En el marco de este proyecto se van a considerar alguna de las opciones software que implementan paravirtualización tanto nativa como albergada.

3.3.3 Virtualización a nivel de sistema operativo

La virtualización a nivel de sistema operativo es una técnica de virtualización en la cual el núcleo del sistema operativo permite la existencia de diferentes instancias (sistemas) independendientes residentes en espacio-de-usuario. Estas instancias reciben el nombre de “entornos virtuales (VEs)” o “contenedores”.

Estos contenedores no virtualizan hardware (ver figura 8), en lugar de ello el núcleo proporciona el soporte necesario para simular a través de aislamiento cada uno de estos entornos virtualizados. Aunque existan múltiples instancias ejecutándose bajo un mismo núcleo, cada una de ellas tiene su propio sistemas de ficheros, memoria, dispositivos, procesos, etc. En muchos casos, un usuario no podría distinguir entre encontrarse utilizando un sistema paravirtualizado o un contenedor pero el proceso de instalación y gestión desde el punto de vista de un administrador de sistemas es sensiblemente diferente.

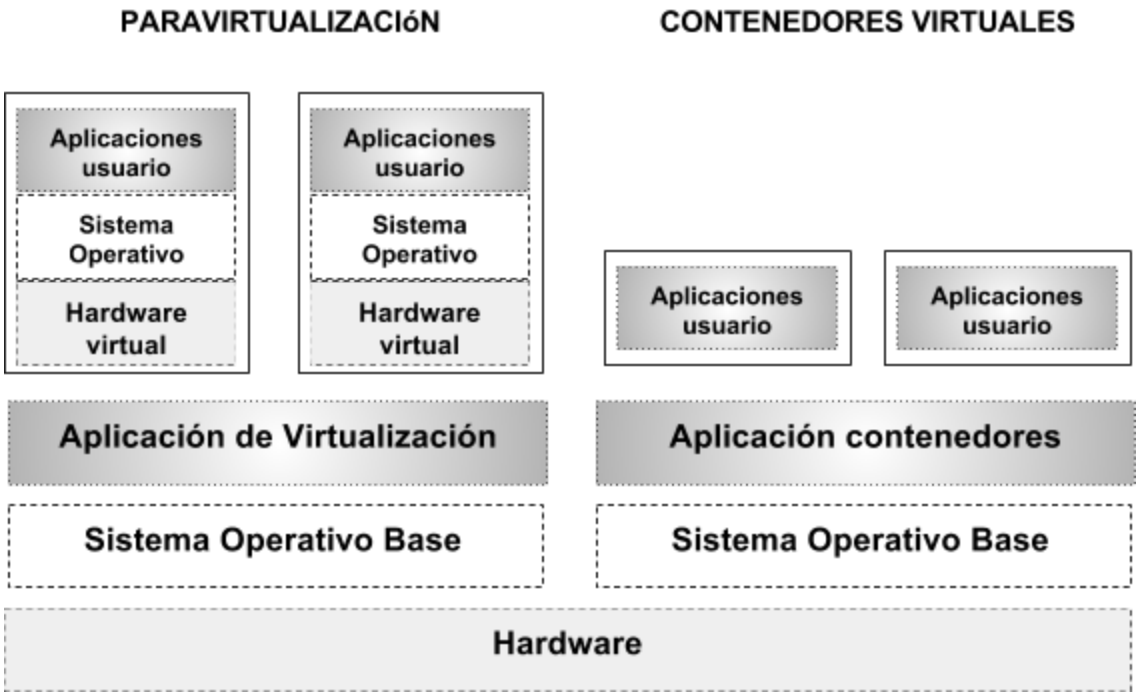


Figura 8: comparación de capas lógicas de sistemas paravirtualizados y contenedores

La historia de la virtualización a nivel de sistema operativo comienza a a finales de los años 70 cuando la llamada de sistema “chroot” fue incorporada a la versión 7 de UNIX y pocos años después a BSD. En 2005 esta funcionalidad alcanzó el kernel de Solaris y en 2008 se crea LXC (Linux Containers project).

Paravirtualización (hypervisor)	Nativa	KVM, Xen, VMWare ESXi etc.
	Albergada	QEMU, Virtualbox, VMWare Player, etc.
Virtualización a nivel de sistema operativo	Docker, chroot, BSD Jails, LXC, ...	

Tabla 1: categorización de sistemas de virtualización y ejemplos

3.3.3.1 Chroot

chroot es un comando Linux que proporciona lo que en la bibliografía [11] de sistema operativos se conoce muchas veces como "jaula" que es un mecanismo de software que proporciona acceso limitado a los recursos del sistema base. El efecto que produce un entorno chroot es el de cambiar la raíz del sistema ("/") por un directorio determinado del sistema. Una ejemplo típico de empleo de chroot se podría resumir en:

1. Preparar el directorio que servirá para cambiar la raíz del sistema con todo el software necesario para replicar un sistema completo o un servicio específico junto con sus dependencias.
2. Cambiar la raíz empleando el comando chroot al directorio creado en el punto 1.
3. Interactuar como root en el sistema creado en el directorio auxiliar.

Por diseño el chroot está principalmente pensado para restringir el acceso al sistema de ficheros [12]. No es posible limitar el acceso a otros recursos de la máquina, cómo el sistema de red o los procesos fuera del entorno del chroot.

Se han empleado históricamente en entornos de pruebas ya que permiten instalar software, compilar o ejecutar pruebas en un entorno separado del propio sistema operativo instalado sobre un hardware real, que efectúa de anfitrión para estos entornos chroot y cuyo sistema de ficheros permanece inalterado durante el empleo de los mismos.

3.3.3.2 LXC - Linux containers

LXC es uno de los proyectos bajo el abanico de Linux Containers[13] que emplea varias características del propio núcleo de Linux para proporcionar una virtualización a nivel de sistema operativo.

El sistema de contenedores emplea la funcionalidad del núcleo de Linux llamada cgroups (control groups) que permite establecer límites y **aislar el empleo de diferentes recursos de un sistema**, como pueden ser el empleo de la CPU, la memoria RAM, el acceso a disco, etc de un grupo de procesos. Además LXC emplea también otra funcionalidad del núcleo conocida como "isolated namespaces" (aislamiento de nombres) que permite una separación completa

de un proceso sobre el resto del sistema operativo incluyendo: otros procesos, usuarios existentes, etc.

LXC proporciona un mejor aislamiento que chroot con una cagar al sistema muy pequeña en términos de empleo de memoria RAM.

3.3.3.3 Docker

El proyecto Docker nace en el año 2013 y supone una gran revolución en la generalización del empleo de contenedores en todos los aspectos del mundo del software. Inicialmente desarrollado sobre Linux, utilizó directamente el proyecto LXC proporcionando una **interfaz mucho más simplificada** que éste.

La otra gran característica que proporcionó el proyecto Docker desde su aparición fue la **memoria caché** que implementa. Basada en sistemas de ficheros multicapa (aufs es el empleado por defecto en Linux, pero se pueden emplear otros como brtfs o zfs) permite almacenar cada uno de los pasos de un sistema en construcción. Por ejemplo: se puede partir de un sistema básico Ubuntu predefinido (se descargará la primera vez y se almacenará en caché), instalar un servidor apache (docker almacenará esté sistema Ubuntu+Apache en su caché) , instalar un servidor mysql (docker almacenará Ubuntu+Apache+Mysql) (ver figura 9).

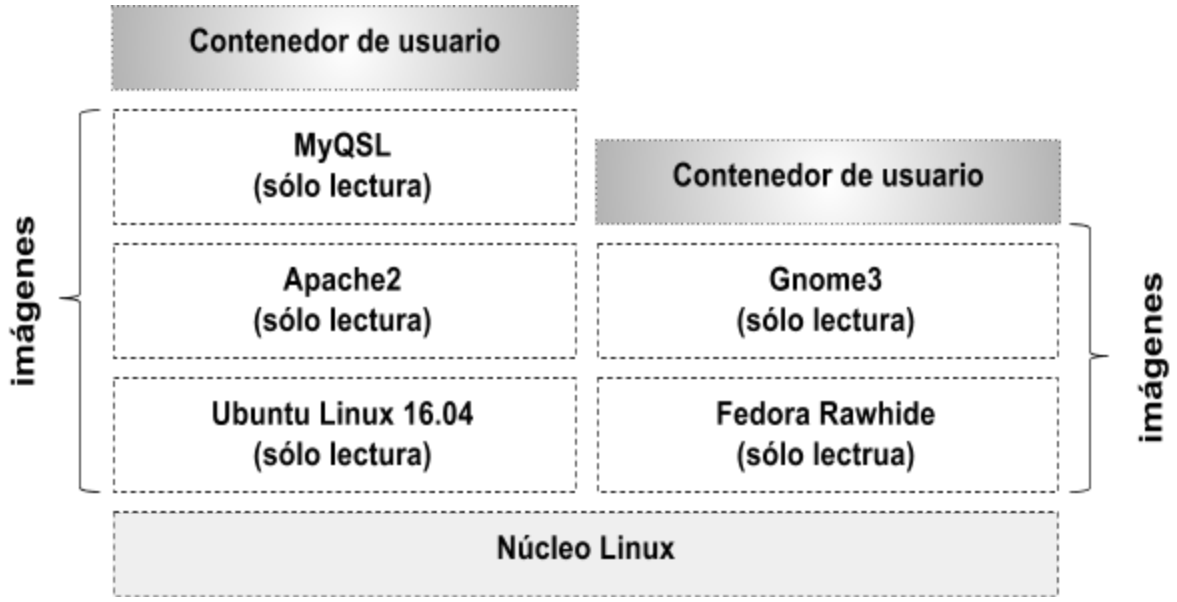


Figura 9: representación de capas en Docker para creación de un sistema LAMP sobre Ubuntu y un entorno de escritorio Gnome3 sobre Fedora

Todos estos almacenamientos de sistemas intermedios o finales en creación o modificación permiten que si el usuario vuelve a repetir los mismo pasos estos sean directamente servidos por la caché de Docker en lugar de tener que esperar a ejecutar cada uno de los pasos en el sistema virtual.

Otra de las grandes ventajas que ofrece el proyecto Docker es una colección pública de sistema pre configurados listos para ser utilizados llamada **Docker Hub**. Es un almacén de todo tipo de sistemas que el usuario puede invocar a través de las diferentes interfaces de docker para que la aplicación se conecte remotamente al Docker Hub y obtenga de él el código necesario para poder ejecutar localmente el sistema requerido.

Capítulo 4

4. **Subproyecto I**: mejora y control del código de generación de los entornos de pruebas.

Este capítulo aborda el análisis, diseño y ejecución del primero de los subproyectos de mejora que tiene relación con la optimización de la gestión del código de generación de entornos de prueba. Se subdivide en descripción (breve reseña sobre el subproyecto en general), objetivo (metas que se propone el subproyecto), estado inicial (situación en el momento de abordar el subproyecto), soluciones existentes (estado de la cuestión sobre el área tratada en el subproyecto) , solución propuesta (solución elegida y su razonamiento) y conclusiones (corolario final sobre el proceso completo).

4.1 Descripción

Este primer subproyecto trata de solucionar el problema existente con el código (scripts) que genera el entorno de pruebas y su ejecución, ya que presenta diversos problemas: está duplicado en cada uno de los trabajos de jenkins, sólo se puede editar mediante la interfaz web del servidor, la propagación de cambios es un proceso manual tendente al error, etc.

4.2 Objetivo

Mejorar el mantenimiento, el desarrollo y la trazabilidad del código que genera el entorno de pruebas y ejecuta las mismas.

4.3 Estado inicial

La primera vez que se configura un trabajo, especialmente si la configuración no va a resultar en exceso compleja, la documentación oficial invita al usuario a utilizar la interfaz web para la introducción de los comandos necesarios para la ejecución de las pruebas.

En el caso de uso que analiza este documento, esta generación del entorno de pruebas consistía en los comandos de bash necesarios para la creación de un entorno virtual (chroot) donde ejecutar posteriormente la compilación y la ejecución de los tests automáticos. Este código se encontraba en el correspondiente formulario web de configuración del propio trabajo (ver figura 10), tal como recomienda la propia documentación.

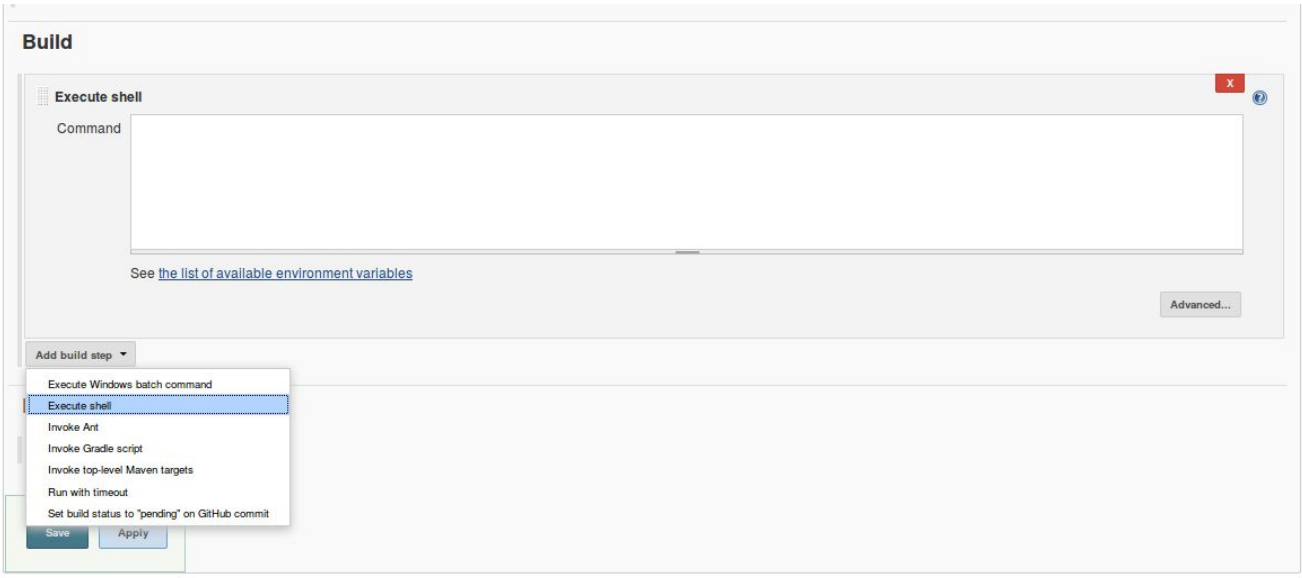


Figura 10: configuración de un trabajo en Jenkins: añadir ejecución de un comando en la terminal

Una vez se guardan los cambios, el nodo master del servidor Jenkins almacena este código junto con el resto de configuración del trabajo en formato XML en la ruta definida por la variable de entorno: */var/lib/job/<nombre_del_trabajo>/config.xml*

4.3.1 Problemas

- **P1** Histórico de cambios: en el escenario inicial, la edición de los comandos *bash* que configuran la creación y ejecución del entorno de pruebas se realiza directamente en el formulario web, para posteriormente almacenar estos datos en formato XML en un fichero del servidor Jenkins. Esto hace que la capacidad de visualizar o analizar los cambios que se van ejecutando sea prácticamente nula. No es posible saber quién, cuándo y qué aspectos del código se ha ido modificando a lo largo del tiempo.

- **P2** Duplicidad de código: la configuración de cada trabajo en Jenkins se almacena en un fichero de configuración XML diferente. Como se ha detallado al principio de esta sección, la configuración está compuesta por diferentes scripts. Con esta organización existe el problema de que el código almacenado en los diferentes archivos de configuración sea muy similar entre sí (comandos que configuración el entorno de virtual de pruebas son idénticos, procedimientos de configuración y compilación son muy similares, etc.). La historia del desarrollo del software [14] ha mostrado en muchas ocasiones la problemática de mantener múltiples versiones de un mismo código: propagación de una solución a un error en el código, necesidad de parchear múltiples copias en presencia de un agujero de seguridad, espacio ocupado por las copias, etc.

- **P3** Respuesta ante contingencias: la instalación del servidor Jenkins del que se disponía en la situación inicial que analiza este trabajo, no implementa ningún procedimiento de copias de seguridad sobre la configuración de los diferentes trabajos de Jenkins. En caso de una incidencia (problemas con el almacenamiento físico o errores de sobreescritura por parte del administrador), el riesgo potencial de perder todas las configuraciones es elevado.

- **P4** Propagación de actualizaciones: aunque podría incluirse en los aspectos cubiertos por anteriormente en el punto P2 (duplicidad de código), figura como punto individual para destacar la importancia del problema principal que afectaba al código a la hora de proceder a su mejora: el desarrollo del mismo. Al implementar un cambio que producía una mejora sobre uno de los trabajos de Jenkins, el mecanismo de propagación del cambio entre los diferentes trabajos era la edición manual de todos aquellos trabajos que compartían el código mejorado, con la dificultad añadida de tener que detectar qué trabajos son los que efectivamente utilizan las mismas instrucciones.

4.4 Soluciones existentes

Siendo una de las principales características de Jenkins su alto grado de personalización y la gran cantidad de extensiones existentes que son capaces de añadir múltiples funcionalidades al servidor, no es de extrañar que las soluciones a muchos de los problemas descritos puedan ser resueltas, o al menos mitigadas, por el empleo de plugins existentes. Analizando punto por punto los problemas que genera la situación inicial descrita:

Falta de un histórico de cambios (P1): para solucionar el problema de no disponer de un registro histórico de modificaciones sobre el código que se encarga de configurar el entorno de pruebas y ejecutarlas, la comunidad de desarrolladores y usuarios de Jenkins desarrolló un plugin que permite disponer de un registro de cambios sobre la configuración de cada uno de los trabajos. El plugin tiene por nombre JobConfigHistory:

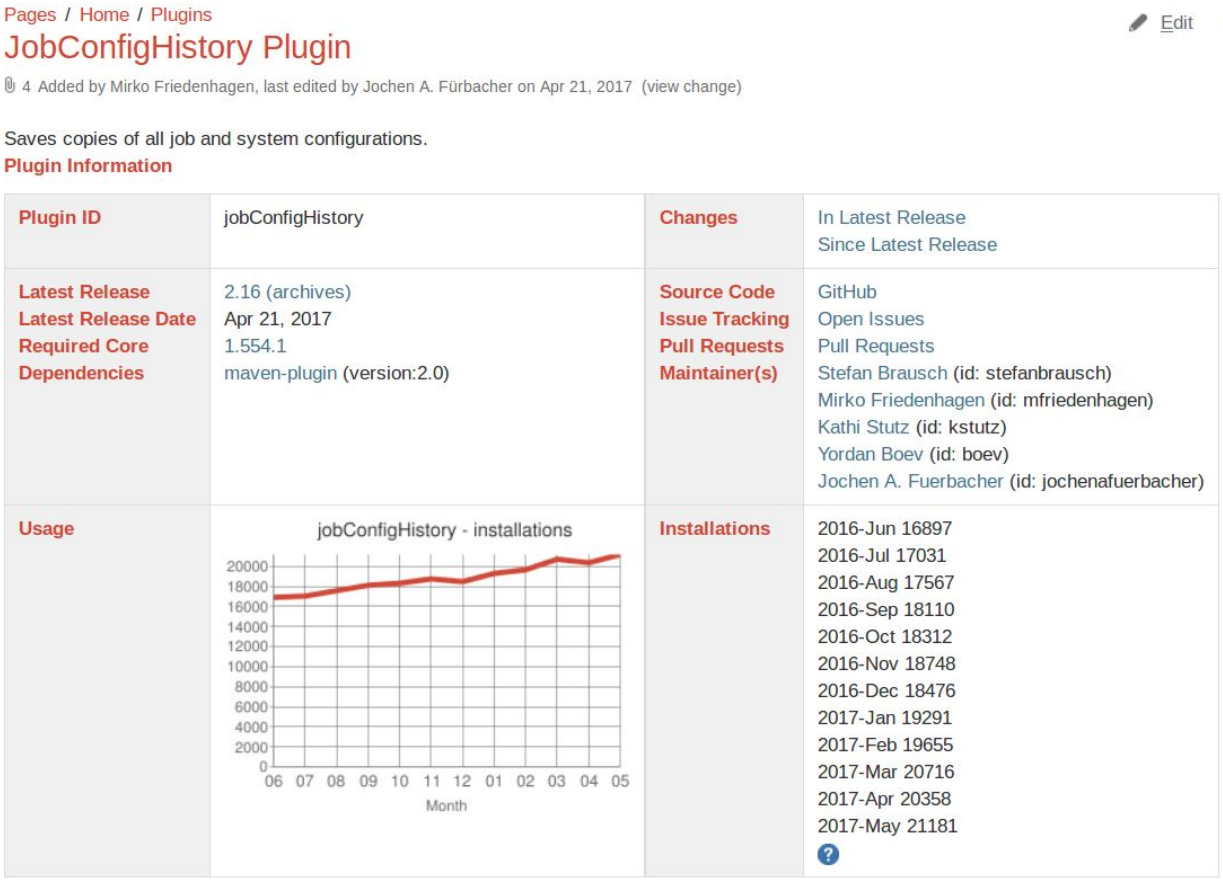


Figura 11: Información sobre la extensión de Jenkins JobConfigHistory

El plugin permite la integración dentro del propio servidor de una herramienta que facilita la visualización de los cambios que se realizan en la configuración:

Figura 12: ejemplo del resultado del plugin de Jenkins JobConfigHistory ante un cambio en la configuración

Como contrapartida, cada plugin que se añade al servidor Jenkins incrementa la memoria que utiliza el proceso y ralentiza su arranque. Otro problema de mayor envergadura es la incapacidad del plugin de revertir los cambios y devolver la configuración y el código a un determinado momento de su desarrollo.

P2 Duplicidad de código: la existencia de código duplicado no es trivial de evitar ya que forma parte de la configuración de cada trabajo y estas configuraciones no son fácilmente intercambiables o compartibles. Para abordar la eliminación de código duplicado habría que replanificar el diseño de los trabajos de Jenkins ya que la disminución de código duplicado va ligada a la disminución del número de trabajos mientras la configuración se almacene empleando la manera predeterminada.

Se podría llegar a reducir el número de trabajos separados por versiones de Ubuntu (ej: gazebo-default-precise y gazebo-default-quantal) y convertirlos en un único trabajo (ej: gazebo-default-generic) que admita un parámetro cuando se invoca y le indique qué distribución se quiere probar.

P3 Respuesta ante contingencias: como contingencias se entiende un abanico de sucesos imprevistos que van desde una acción errónea con un efecto leve, como puede ser una configuración incorrecta, hasta un accidente que ocurre en el servidor en el que se ejecuta Jenkins y afecta al disco duro, como puede ser una corrupción fatal en el sistema de ficheros.

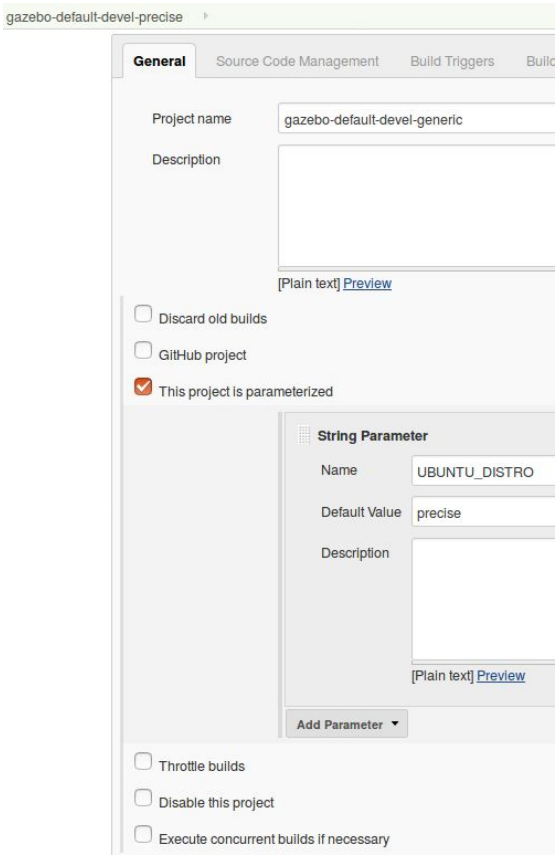


Figura 13: Parametrización de un trabajo en Jenkins

Con el sistema inicial, almacenando la configuración de cada trabajo de Jenkins en ficheros, XML albergados típicamente bajo */var/jenkins*, cualquier incidente relacionado con la pérdida o alteración de los mismos genera un problema muy grave al tener que volver a configurar todos los trabajos uno por uno con la interfaz gráfica.

Existen al menos dos aproximaciones para solucionar el problema:

1. La primera sería utilizar herramientas de backup (e.j. Bacula) o sincronización (e.j. rsync, lsync, …) desde el propio servidor Linux que alberga Jenkins para disponer de un sistema que permita recuperar los archivos o un estado previo de los mismos.
2. La segunda sería el empleo de **herramientas nativas de Jenkins** para realizar la misma labor de copia de seguridad. En la primera aproximación habría que sincronizar el estado del servicio de Jenkins con las copias de seguridad para evitar posibles casos de corrupción por escritura/lectura paralela. También como punto a favor de emplear la segunda opción, herramienta nativa Jenkins, se puede destacar que las herramientas de gestión de contingencias nativas proveen generalmente de opciones que van más allá de salvaguardar solamente la propia configuración de los trabajos y alcanzan otros aspectos también importantes de la aplicación como pueden el respaldo de los registro de uso (logs) de cada ejecución.

La instalación estándar de Jenkins no ofrece ningún servicio de copia de seguridad pero nuevamente su configurabilidad y una comunidad activa ofrecen distintas soluciones en forma de plugin. El que tiene mayor número de instalaciones y está activamente desarrollado se conoce como *thinBackup*

.

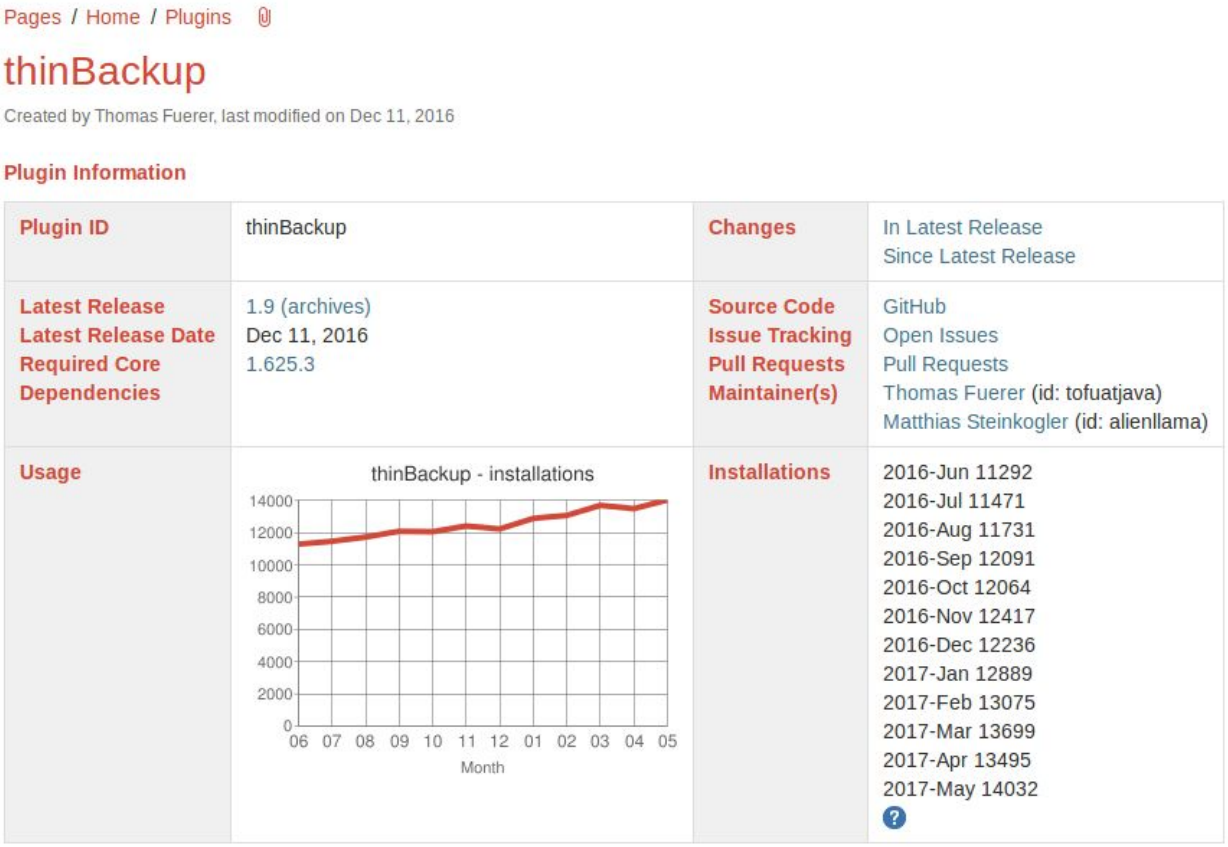


Figura 13: Información sobre la extensión de Jenkins thinBackup

La extensión *thinBackup* permite configurar fácilmente la generación de copias de respaldo, se integra perfectamente con la actividad del propio Jenkins (espera a que terminen los trabajos en ejecución para realizar la copia) y ofrece una copia completa de todos los archivos que se encuentran en el directorio raíz de Jenkins (no solamente la configuración de cada trabajo) sino otros datos como los plugins instalados, la última ejecución exitosa, los registros de los propios trabajos, el número de ejecuciones realizadas por cada trabajo, etc.

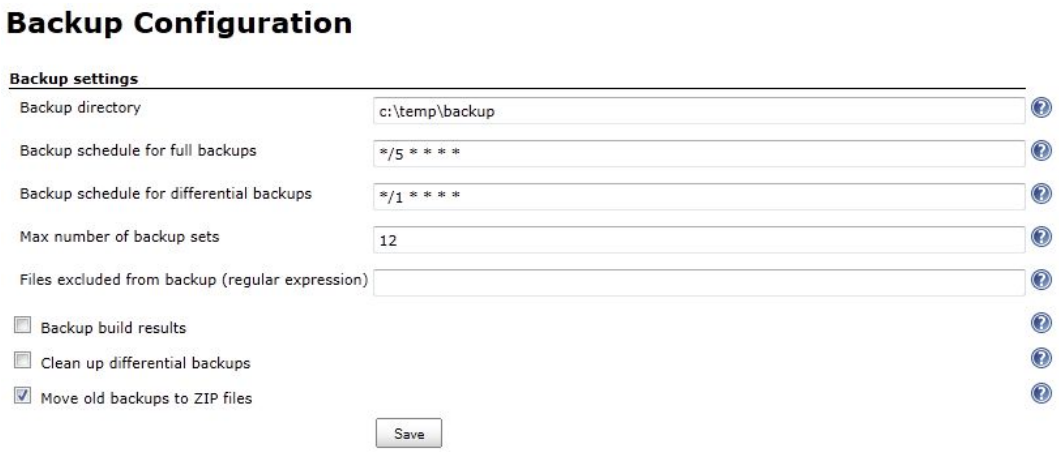


Figura 14: formulario de configuración de la extensión de Jenkins ThinBackup

4.5 Solución propuesta

Uno de los principales problemas que tenía la configuración inicial era la duplicidad de código. Muchos de los trabajos de Jenkins utilizaban código muy similar. En algunos casos era exactamente igual y solamente cambiaba el nombre de la distribución de Ubuntu. Por ejemplo: trabajos de pruebas para Ubuntu Precise y Ubuntu Quantal que se crean simplemente copiando la configuración y cambiando el nombre de la distribución.

Una posible solución sería **alojar el código de generación y ejecución de test en un único lugar** y hacer que todos los trabajos de Jenkins lo obtuvieran de este único punto. Existen diversas soluciones técnicas que podrían cumplir con esta condición, como pueden ser la descarga de un tarball con el código o el empleo de un repositorio de código.

Teniendo en cuenta la lista de los problemas a resolver en este punto, el empleo de un repositorio de código hace que no solamente la duplicidad de código quede resuelta sino que también, al emplear un control de versiones, tanto la trazabilidad de los cambios como la propagación de las actualizaciones sean resueltas directamente.

Un sistema de control de versiones es una herramienta idónea para asegurar la trazabilidad de los cambios en un código fuente, siendo una solución extendida y utilizada al menos desde la aparición de CVS, en la década de los años noventa, y se generalizaron con la aparición de sitios web de alojamiento de repositorios gratuitos (como Github o Bitbucket).

Para solucionar el problema de cómo propagar las actualizaciones sobre el código, si disponemos del mismo almacenado en un repositorio, la configuración de los trabajos se debería ajustar para conseguir que cada vez que un trabajo de Jenkins se ejecute, lo primero que se haga es una comprobación y descarga de los últimos cambios existentes en el repositorio (ver figura 15). De esta manera, el repositorio ejercería como el único punto de cambio y las actualizaciones del código de preparación y ejecución de pruebas se propagarían automáticamente cuando los distintos trabajos de Jenkins sean iniciados.

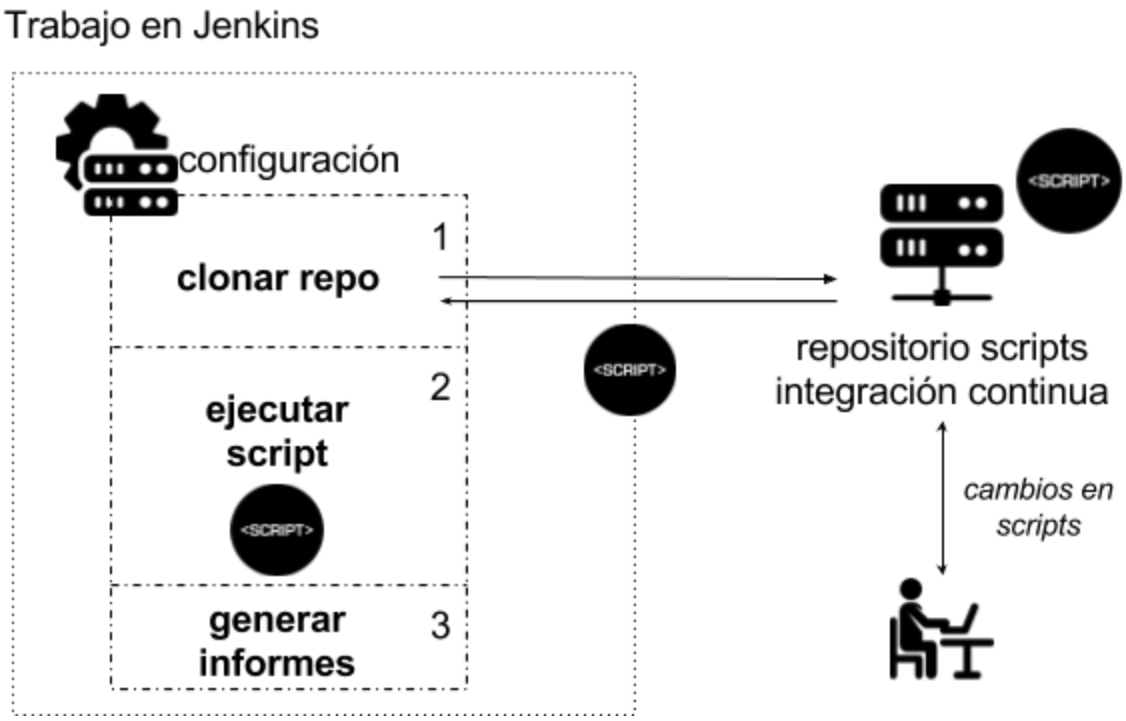


Figura 15: proceso de obtención de los comandos para el proceso de pruebas automáticas desde la configuración de un trabajo en Jenkins

En resumen: muchas de las soluciones evaluadas mitigan problemas descritos en el estado inicial pero lo hacen de manera individual sobre cada problema. Probablemente se podría llegar a una solución integral mediante el empleo combinado de varias de las soluciones propuestas. Al evaluar el **empleo de un repositorio de código como solución final se ha encontrado que es capaz de solucionar los problemas de manera estructural y proporcionar otras ventajas adicionales** propias tanto de un sistema de control de versiones (empleo de ramas o etiquetas) como de los sitios web que actualmente soportan (facilitar el peer-review, etc).

De esta manera surgió el repositorio que se ha utilizado durante los últimos 4 años por parte del proyecto Gazebo (y subproyectos del mismo). Toma el nombre de release-tools y está alojado en la cuenta de bitbucket oficial de la que dispone la Open Source Robotics Foundation (OSRF): https://bitbucket.org/osrf/release-tools.

4.6 Conclusión

Se partía de un sistema con numerosos defectos desde el punto de vista de mantenimiento del código y la transición ha permitido solucionar los aspectos que se detallaron en el apartado de problemas.

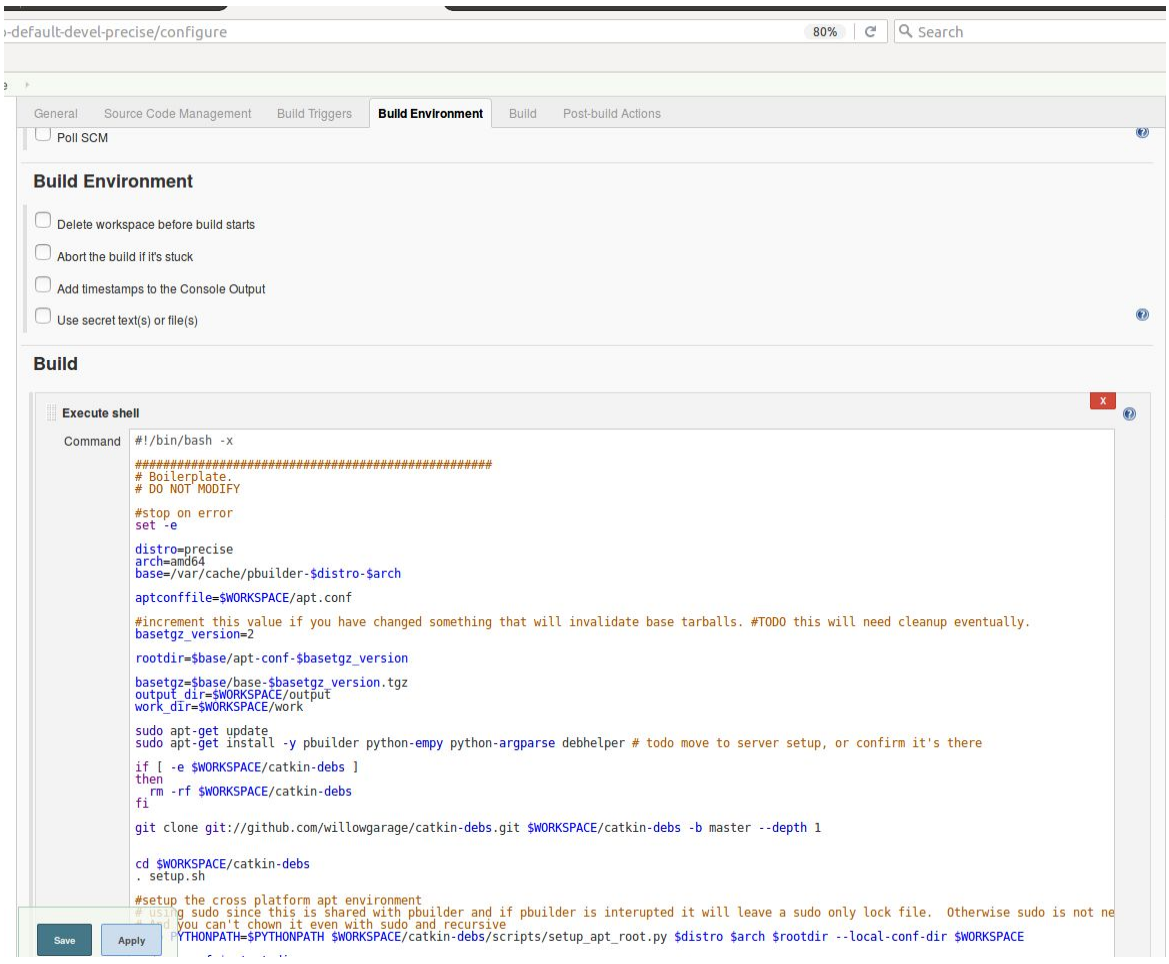
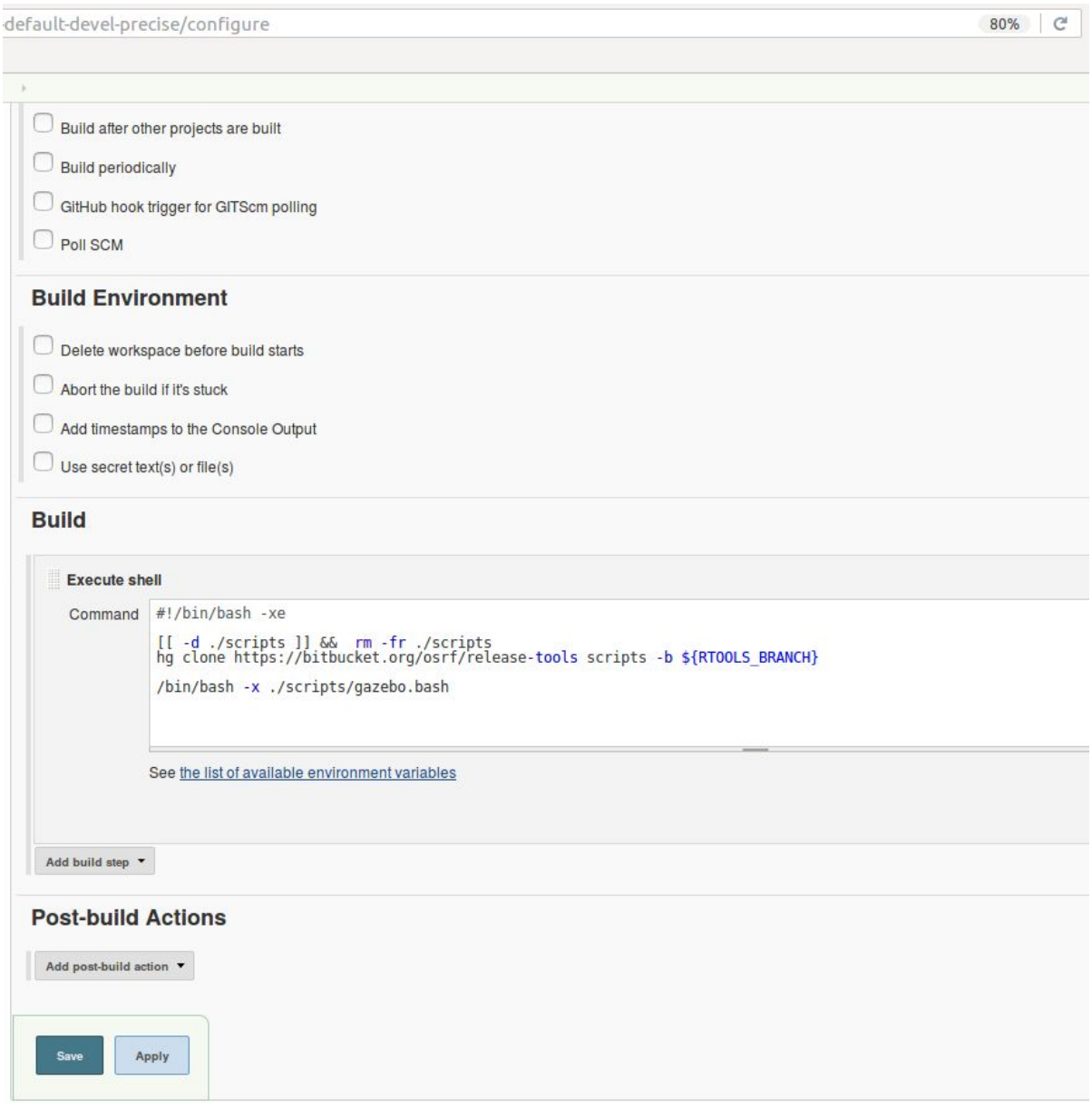


Figura 16: código de ejecución de pruebas automáticas directamente almacenado en la interfaz del trabajo en Jenkins. Estado inicial del subproyecto.

La solución elegida supone un cambio estructural en la metodología de trabajar con el código destinado a generar el entorno de pruebas, al introducir el empleo de un sistema de control de versiones. El hecho de alojar el código en un sistema de control de versiones no sólo soluciona los problemas iniciales descritos sino que abre la puerta a emplear otras posibilidades que ofrecen tanto la propia herramienta de control de versiones, cómo puede ser el uso de ramas o *tags*, así como las ventajas que ofrecen los sitios web que alojan el código: revisión gráfica de pull requests, visualización de cambios en formato web, *forks* automáticos para otros usuarios, etc.



Solución propuesta: código que obtiene una copia del repositorio de comandos para pruebas automáticas y ejecución del script gazebo.bash

Funcionalidad	Situación inicial previa	Solución propuesta
Propagación de un cambio en el código	Requería de realizar N veces, una por cada trabajo en el servidor de CI. Necesaria interacción con el servidor via interfaz web. No escalable.	Una sola modificación en el sistema de control de versiones. El cambio se propaga automáticamente cuando un trabajo es ejecutado.
Consulta del histórico de cambios en la configuración	No es posible	Facilidad para ver los cambios en la web que aloja el código. También se puede utilizar la línea de comandos directamente interactuando con el sistema de control de versiones o incluso alguna de las herramientas gráficas que existen para el VCS.
Deshacer un cambio realizado	No es posible	Es trivial utilizar el VCS que alberga el código para revertir un cambio o volver a versiones antiguas del código. Se puede utilizar la línea de comandos o herramientas gráficas que interactúen con el VCS.
Respuesta ante contingencias (e.j: pérdida del disco del servidor)	Era necesario mantener manualmente alguna copia de seguridad de los scripts y restablecerla implicaba inseguridad sobre lo actualizado de la misma además del problema de tener que volver a configurar trabajo a trabajo todo el servidor.	El código se almacena en un servicio externo que se encarga de su disponibilidad, actuando como un backup permanentemente actualizado. Configurar un nuevo servidor es tan sencillo como establecer las instrucciones de descarga del código del VCS.
Velocidad de ejecución	Inmediata. El código figura en el archivo de configuración del propio servidor de CI.	La obtención del código desde el servidor de VCS demora algunos segundos (dependendiendo del tamaño del histórico, entre 2 y 7 para un repositorio con años de cambios)

Tabla 2: comparativa entre la situación inicial y la solución propuesta de funcionalidades relacionadas con el código de generación de entornos de pruebas

Cabe también destacar que un proceso de análisis y búsqueda de soluciones a diversos problemas que termina encontrado una solución sencilla y eficaz, como la que se habría podido encontrar en este capítulo, resulta especialmente valiosa. Ante los numerosos problemas existentes inicialmente, tiene relevancia el encontrar una **solución que evita la sobreingeniería**.

El efecto adverso que supone la implantación del sistema de control de versiones es el tiempo que tarda en realizarse una obtención del código desde el servidor. Este tipo de transferencias se han ido optimizado por parte de los diferentes sistemas de control de versiones y a día de hoy un repositorio con un histórico de cambios relativamente pequeño, sin necesidad de unas condiciones de ancho de banda potentes, no tarda más de un par de segundos.

En casos en que ese repositorio evoluciona y guarda un histórico mayor, podemos ver que el tiempo se incrementa sin suponer un problema para ejecuciones de testing que duran varias decenas de minutos. Esta contrapartida es asumible teniendo en cuenta que la implementación es capaz de solucionar toda la problemática existente inicialmente.

Capítulo 5

5. **Subproyecto II**: soporte para la ejecución de pruebas automáticas que requieran aceleración gráfica

Este capítulo aborda el análisis, diseño y ejecución del segundo de los subproyectos de mejora que tiene relación con el soporte para el acceso al hardware de aceleración gráfica desde el entorno virtual de pruebas. Se subdivide en descripción (breve reseña sobre el subproyecto en general), objetivo (metas que se propone el subproyecto), estado inicial (situación en el momento de abordar el subproyecto), soluciones existentes (estado de la cuestión sobre el área tratada en el subproyecto) , solución propuesta (solución elegida y su razonamiento) y conclusiones (corolario final sobre el proceso completo).

5.1 Descripción

Este apartado aborda la falta de soporte del entorno virtual de testing para la ejecución de las pruebas automáticas que requieran de aceleración gráfica. Algunos componentes del simulador Gazebo utilizan la aceleración gráfica para optimizar su funcionamiento, especialmente en la simulación de determinados sensores. El entorno en el que se ejecutan las pruebas automáticas no debería obstaculizar el acceso al hardware de aceleración para poder ejecutar los tests que requieren del empleo de este hardware.

5.2 Objetivo

Implementar el soporte necesario para que el entorno de testing puede acceder correctamente al hardware de aceleración gráfica y ejecutar el conjunto completo de pruebas automáticas de las que dispone el código. Como objetivo secundario, mientras dura la fase de implementación fijada como objetivo principal, deshabilitar los tests que requieran aceleración gráfica.

5.3 Estado inicial

Los entornos virtuales iniciales empleados para la ejecución tests estaban basados en el empleo de chroots sobre una plataforma Ubuntu Linux. El servidor Jenkins disponía de varios agentes de ejecución con sistema base también bajo Ubuntu Linux y un hardware de aceleración gráfica, tarjetas *Nvidia Geforce GTX 750*. El sistema base ejecuta un servidor gráfico con aceleración hardware utilizando los drivers propietarios para Ubuntu que proporciona la propia empresa fabricante.

Aunque el sistema que sirve de base al nodo de Jenkins dispone del hardware y software de aceleración gráfica adecuado, los entornos de pruebas creados por **chroot eran incapaces de acceder a la aceleración** (ver figura 17), se producía un error del tipo segfault.

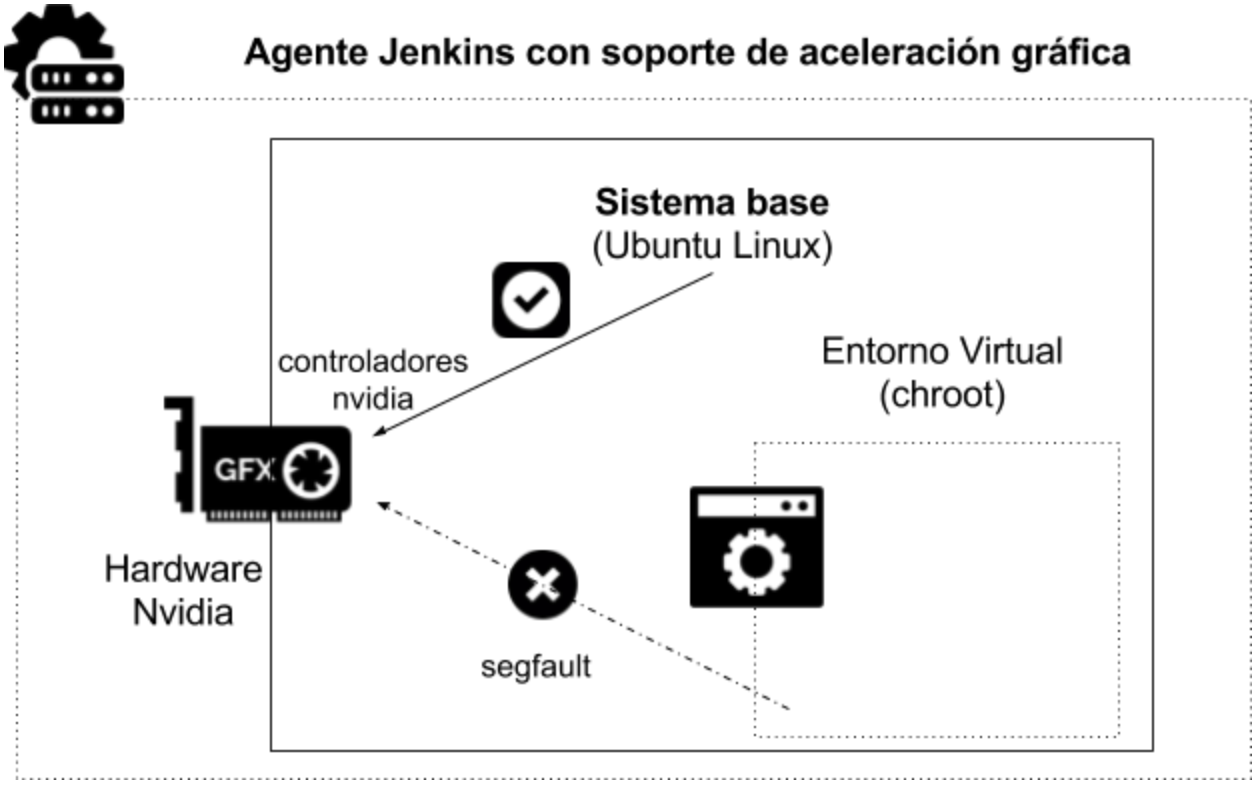


Figura 17: sistemas existentes en un agente Jenkins con acceso
al hardware de aceleración gráfica

Los entornos chroot sí que podían emplear con normalidad el servidor gráfico (sin aceleración hardware) si el sistema base no utilizaba los drivers propietarios de Nvidia y en su lugar se empleaba el driver de código abierto (*Nouveau*) o el chip Intel integrado en la placa base. Esta situación inicial hacía que los tests que requerían de aceleración gráfica fallaran continuamente en los entornos de pruebas chroot disponibles.

5.4 Soluciones existentes

Para disponer de un servidor gráfico con la capacidad de aceleración gráfica en un entorno de pruebas virtual, existen al menos dos aproximaciones principales: la primera consiste en, como trataba de implementar el estado inicial de partida del proyecto, emplear el propio servidor que ejecuta el sistema anfitrión del nodo de Jenkins, también encargado de crear los propios entornos virtuales. La segunda de las posibles soluciones sería el empleo por parte de los entornos chroot de su propio servidor X virtual.

5.4.1 Acceso al servidor la aceleración gráfica desde chroot

Un chroot es un mecanismo de virtualización con un grado de aislamiento bajo en el cual hay directorios del sistema base que son accesibles directamente desde el sistema chroot, como por ejemplo /dev, /proc o /sys. Con este diseño, un chroot podría acceder directamente a las interfaces creadas por los drivers de las distintas tarjetas gráficas. Por ejemplo, en el caso de NVIDIA (bajo un núcleo 4.4.81): */dev/nvidia* , /proc/driver/nvidia*, /sys/module/nvidia.*

El poder acceder a estos drivers directamente en el sistema de ficheros resulta insuficiente para disponer de las funcionalidades de aceleración gráfica. Se necesita que el chroot disponga de las mismas librerías y software que implementa la aceleración gráfica en el sistema base [15], de tal manera que la implementación del rendering y la interacción con el driver del núcleo de Linux sea la esperada (es el trabajo que realiza una distribución Linux cuando prueba las diferentes versiones de cada proyecto software para que interactúen correctamente entre ellas). Para conseguir esto se debería instalar en el **chroot las mismas versiones de drivers que tiene el sistema base** (ver figura 18).

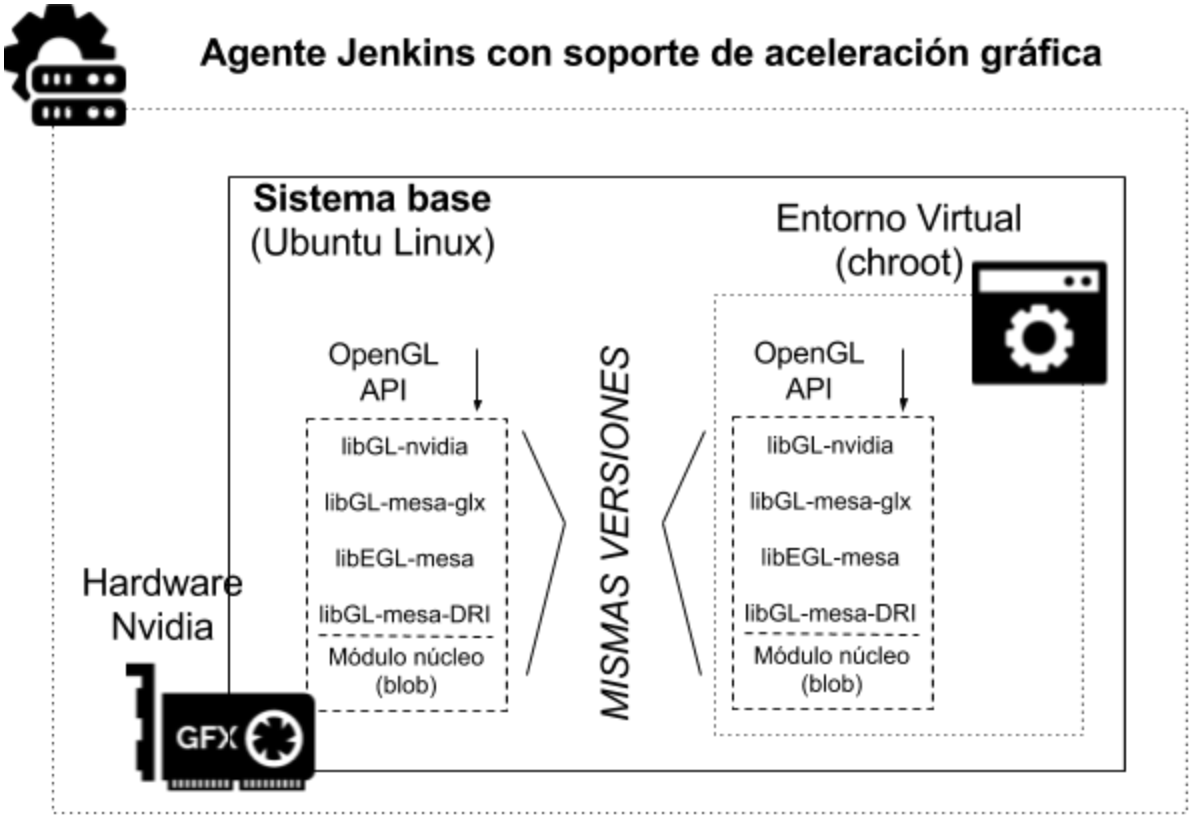


Figura 18: sincronización de software necesaria para acceder al hardware de aceleración gráfica por parte del sistema base y el entorno virtual

5.4.2 Servidores X virtuales

Otra aproximación consistiría en que cada entorno de testing pudiera lanzar su propio servidor gráfico. Al tener entornos de testing 'virtualizados' que no disponen de acceso a hardware se necesita encontrar alguna implementación de un servidor gráfico virtual o la virtualización del hardware.

Xvfb (X virtual framebuffer) forma parte del proyecto Xorg y es quizá el proyecto más longevo que implementa un servidor gráfico virtualizado en entornos Linux. Implementa el protocolo X11, incluidos los eventos propios de servidor real, pero emplea memoria virtual para realizar las operaciones gráficas. Su empleo está principalmente focalizado en los entornos de pruebas y es realmente sencillo de utilizar. Para lanzar un servidor en el número 1 y pantalla número 0 con una resolución de 1600x1200 y una profundidad de 32 bits bastaría:

```
Xvfb :1 -screen 0 1600x1200x32
```

Al ser una representación virtual de un framebuffer no soporta aceleración software o hardware.

Xnest, Xephyr o Xdummy (sustituye a Xephyr) son implementaciones basadas en un modelo que utiliza una ventana de un servidor existente como su framebuffer. Xnest fue la primera de ellas y no soportaba el empleo de extensiones X. Xephyr es una evolución sobre Xnest que sí es capaz de implementar extensiones X (*Composite*, *randr*, etc.) pero no dispone de soporte para aceleraciones GLX (OpenGL extension for X server). Xdummy sí que dispone de extensiones GLX para aceleración gráfica y es la última de las evoluciones de este tipo de proyectos que se podría aproximar como a un servidor *X-dentro-de-X*.

5.5 Solución propuesta

Al evaluar las opciones que implementan servidores gráficos virtuales, en concreto Xdummy que es capaz de soportar aceleración gráfica, apareció uno de los problemas fundamentales: el rendimiento de aceleración gráfica.

Al ser Gazebo un simulador 3D de escenarios virtuales, la demanda de renderización en tres dimensiones es alta, lo podríamos aproximar a las necesidades de un juego shooter en 3D como pudiera ser Unreal. Especialmente en entornos complejos, la aceleración software o el empleo de drivers open source (como *Nouveau*) es muy probable que sea insuficiente para obtener una tasa de imágenes por segundo razonable.

Con esta limitación se procedió a intentar utilizar directamente el servidor X existente en el sistema host, controlado por el driver propietario de NVIDIA y por tanto con la aceleración por hardware más alta de entre las opciones existentes.

Se aborda en primer lugar el objetivo secundario, deshabilitar los tests de aceleración gráfica en aquellas máquinas que no disponen de soporte para los mismos. Este proceso está compuesto de dos fases: por un lado, implementar el soporte necesario para la detección de la aceleración gráfica y por otro dotar al sistema de configuración/compilación de Gazebo (cmake) del soporte para el marcado y selección de las pruebas que requieren de este tipo de soporte.

5.5.1 Detección de soporte de aceleración gráfica

En un primer momento se aborda la implementación de un módulo en cmake que sea capaz de reportar si el sistema en el cual se está ejecutando la compilación dispone de aceleración gráfica. Como se detalla en el punto 3.1 (estado de la cuestión, Gazebo) el simulador utiliza para la renderización 3D la libería OpenGL (Open Graphics Library). En sistemas UNIX que emplean el servidor X, una de las extensiones del servidor que interactúa con OpenGL es GLX (*OpenGL Extension to the X Window System*).

Uno de los componentes que existe dentro de las implementaciones open source de OpenGL es MESA. El proyecto MESA facilita una serie de herramientas que permite interactuar con los sistemas Linux y conocer el estado del soporte del sistema con respecto a OpenGL. Este conjunto de herramientas recibe el nombre de *mesa-utils* y dentro del conjunto de ejecutables que proporciona este proyecto, encontramos glxinfo que muestra información sobre la implementación GLX de un servidor X en ejecución.

La opción de utilizar esta herramienta y crear un módulo en cmake que extraiga el resultado relevante y proporcione la información al sistema de configuración es una de las primeras opciones que están disponibles. Una posible implementación de esta idea en cmake sería:

```
MESSAGE(STATUS "Looking for a valid DRI display")
SET (VALID_DRI_DISPLAY FALSE)

# Try to run glxinfo. If not found, variable will be empty
FIND_PROGRAM(GLXINFO glxinfo)

# If not display found, it will throw an error
# Another grep pattern: "direct rendering:[[:space:]]*Yes[[:space:]]*"
IF (GLXINFO)
  EXECUTE_PROCESS(
    COMMAND glxinfo
    COMMAND grep GL_EXT_framebuffer_object
    ERROR_QUIET
    OUTPUT_VARIABLE GLX)

  IF (GLX)
    MESSAGE(STATUS " + found a valid dri display (glxinfo)")
    SET (VALID_DRI_DISPLAY TRUE)
  ENDIF ()
ENDIF()
```

Esta aproximación a la resolución del problema presenta aspectos negativos que pueden hacerla frágil. Si no se dispone del ejecutable glxinfo, que no figura como núcleo mínimo de utilidades en la mayor parte de las distribuciones Linux, el módulo de detección, evidentemente, fracasa. También presenta algunos problemas de portabilidad, no está disponible en todas las plataformas especialmente en Win32.

Otro método alternativo sería utilizar alguno de los múltiples lenguajes que implementan bindings sobre OpenGL. Uno de los más utilizados es python y sus bindings *python-opengl*. Si el sistema dispone de estos bindings que utilizan otras aplicaciones se podrían utilizar para conocer el estado de la aceleración. Para ello se implementó un pequeño script de testing que emplean python-opengl y genera una salida suficientemente descriptiva para conocer el estado de GLX:

```
#!/usr/bin/env python
from OpenGL.GLUT import *
import sys

glutInit(sys.argv)

# Select type of Display mode:
# Double buffer
# RGBA color
# Depth buffer
# Alpha blending
glutInitDisplayMode(GLUT_RGBA | GLUT_DOUBLE | GLUT_DEPTH | GLUT_ALPHA)

# get a 640 x 480 window
glutInitWindowSize(640, 480)

# the window starts at the upper left corner of the screen
glutInitWindowPosition(0, 0)

# Open a window
window = glutCreateWindow("Testing DRI")
```

Desde cmake crear un código que invoque el script y obtenga el resultado es una tarea sencilla:

```
EXECUTE_PROCESS(
     # RESULT_VARIABLE is store in a FAIL variable since the command
     # returns 0 if ok and 1 if error (inverse than cmake IF)
     COMMAND ${PROJECT_SOURCE_DIR}/tools/gl-test.py
     RESULT_VARIABLE GL_FAIL_RESULT
     ERROR_QUIET
     OUTPUT_QUIET)

IF (NOT GL_FAIL_RESULT)
  MESSAGE(STATUS " + found a valid dri display (pyopengl)")
  SET (VALID_DRI_DISPLAY TRUE)
ENDIF ()
```

El resultado final de esta implementación, con algún parche más para algunos casos residuales de errores, figura en el *Anexo 4: módulo cmake detección de soporte OpenGL*.

5.5.2 Implementación tests que necesitan GLX

Inicialmente el proyecto Gazebo disponía un macro auxiliar en cmake que se encargaba de la generación de tests, realizando distintas operaciones a partir del código fuente en C++ y utilizando tanto la librería Google Test como el framework CTest. En versión simplificada:

```
macro (gz_build_tests)
  # Build all the tests
  foreach(GTEST_SOURCE_file ${ARGN})
    string(REGEX REPLACE ".cc" "" BINARY_NAME ${GTEST_SOURCE_file})
    add_executable(${BINARY_NAME} ${GTEST_SOURCE_file})
    include_directories("${PROJECT_SOURCE_DIR}/test/gtest/include")
    add_dependencies(${BINARY_NAME}
      gtest gtest_main
      gazebo_sdf_interface
      gazebo_common
      gazebo_math
...
    add_test(${BINARY_NAME} ${CMAKE_CURRENT_BINARY_DIR}/${BINARY_NAME}
      --gtest_output=xml:${CMAKE_BINARY_DIR}/test_results/${BINARY_NAME}.xml)

    set_tests_properties(${BINARY_NAME} PROPERTIES TIMEOUT 240)

    # Check that the test produced a result and create a failure if it didn't.
    # Guards against crashed and timed out tests.
    add_test(check_${BINARY_NAME} ${PROJECT_SOURCE_DIR}/tools/check_test_ran.py
             ${CMAKE_BINARY_DIR}/test_results/${BINARY_NAME}.xml)
  endforeach()
endmacro()
```

Para los tests que dependen de aceleración gráfica se creo un segundo macro (*gz_build_gpu_tests*) que, dependiendo de si la detección de la GPU había sido satisfactoria era equivalente con el macro standard de creación de tests y si la detección de la GPU había resultado fallida, el macro se definía como vacío.

```
if (VALID_DISPLAY)
  # Redefine build display tests
  macro (gz_build_display_tests)
    gz_build_tests(${ARGV})
  endmacro()
else()
  # Fake macros when no valid display is found
  macro (gz_build_display_tests)
  endmacro()
endif()
```

Para la implementación de la restricción que supone la necesidad de instalar todas las capas software que intervienen en la aceleración gráfica en el sistema chroot, imitando exactamente las versiones disponibles en el sistema anfitrión para así poder acceder a la aceleración gráfica, se realizó como parte de la creación del entorno de pruebas automáticas añadiendo el soporte necesario a los scripts.

Obtener la versión exacta del controlador de aceleración gráfica necesario resultó no ser una tarea trivial. La aproximación de obtener la versión desde el propio sistema de paquetes del sistema host presenta el problema de poder tener instalados en el sistema varias versiones de los controladores al mismo tiempo, aunque solamente una esté en uso por parte del núcleo. Al no ser suficiente la consulta al gestor de paquetes para conocer qué versión del controlador se está ejecutando, es necesario emplear herramientas que interpelen al núcleo en ejecución sobre la información de la que dispone el mismo.

lspci es una utilidad del proyecto pciutils que muestra información detallada sobre todos los buses PCI y los dispositivos asociados a los mismos, como pueden ser las tarjetas gráficas. Revela información sobre los diferentes controladores en ejecución y la versión de los mismos. La idea es utilizar la salida del comando para conseguir las versiones de software que está ejecutando el núcleo del sistema host para poder replicar la instalación en el sistema chroot.

```
GRAPHIC_CARD_FOUND=false
GRAPHIC_CARD_PKG=""

# Check for Nvidia stuff. Using nvidia-docker no installation needed
if [ -n "$(lspci -v | grep nvidia | head -n 2 | grep "Kernel driver in use: nvidia")" ]; then
    export GRAPHIC_CARD_NAME="Nvidia"
    export GRAPHIC_CARD_FOUND=true
fi

# Check for ati stuff
if [ -n "$(lspci -v | grep "ATI" | grep "VGA")" ]; then
    # TODO search for correct version of fglrx
    export GRAPHIC_CARD_PKG=fglrx
    export GRAPHIC_CARD_NAME="ATI"
    export GRAPHIC_CARD_FOUND=true
fi

# Check for intel
if [ -n "$(lspci -v | grep "Kernel driver in use: i[0-9][0-9][0-9]")" ]; then
    export GRAPHIC_CARD_PKG="xserver-xorg-video-intel"
    export GRAPHIC_CARD_NAME="Intel"
    export GRAPHIC_CARD_FOUND=true
    # Need to run properly DRI on intel
    export EXTRA_PACKAGES="${EXTRA_PACKAGES} libgl1-mesa-dri"
fi
```

Existe otro componente necesario para utilizar el servidor gráfico existente desde el entorno chroot, conocer cúal es el socket de comunicación que emplea el servidor Xorg para comunicarse con sus cliente (aplicaciones gráficas). En los sistemas UNIX típicamente se puede conocer con el valor de la variable DISPLAY pero el socket existe en el sistema de ficheros dentro del directorio oculto */tmp/.X11-unix/*.

Además de explorar el contenido de este directorio para fijar la variable DISPLAY se añadió a los scripts de ejecución de pruebas automáticas otro método que consiste en explorar directamente la lista de procesos y buscar el servidor X para obtener qué número de pantalla y secuencia está empleando. El código que implementa esta idea:

```
export_display_variable()
{
    # Hack to found the current display (if available) two steps:
    # Check for /tmp/.X11-unix/ socket and check if the process is running
    for i in $(ls /tmp/.X11-unix/ | sed -e 's@^X@:@')
    do
      # grep can fail so let's disable the fail or error during its call
      set +e
      ps aux | grep bin/X.*$i | grep -v grep
      set -e
      if [ $? -eq 0 ] ; then
        export DISPLAY=$i
      fi
    done
}
```

El script completo puede consultar en el Anexo 4 (código de detección del driver gráfico en uso).

Existía un último obstáculo para poder acceder desde el entorno chroot al servidor de X del sistema host. Al ser un sistema cliente <-> servidor el servidor X tiene medidas de protección básicas sobre qué sistemas tienen acceso al mismo y qué usuarios.

El sistema X en los sistemas base de los nodos de Jenkins era lanzado por el gestor de sesiones lightdm (opción por defecto en los sistemas Ubuntu) bajo un usuario genérico del sistema (de nombre osrf). Para que el proceso y subprocesos del servidor Jenkins (que opera bajo el usuario de sistema jenkins) pueda acceder al servidor X es necesario dotar de autorización al mismo.

Para modificar la autorización de máquinas o usuarios en el servidor X, éste proporciona una aplicación llamada xhost. Un pequeño script que dota de acceso únicamente local a los usuarios root, jenkins y osrf sería:

```
#!/bin/sh
xhost +si:localuser:root
xhost +si:localuser:jenkins
xhost +si:localuser:osrf
```

Sólo faltaría configurar el sistema para emplear el script en cada inicio. Para esta finalidad se utilizó el propio gestor de sesiones lightdm, configurándolo para que al iniciarse ejecutará la autorización necesaria sobre el servidor X mediante el script anterior. Para ello, en el fichero de configuración *lightdm.conf* se configuró la directiva display-setup-script que se encarga de ejecutar un script dado con permisos de root cuandos se inicia el gestor de sesiones.

```
[SeatDefaults]
display-setup-script=/etc/lightdm/xhost.sh1
```

5.6 Conclusión

La falta de aceleración gráfica de los posibles **entornos gráficos virtuales** desaconsejaron su uso ya que el conjunto de pruebas automáticas no quedaría complementamente satisfecho. Con esta opción inhabilitada, el **empleo del servidor gráfico existente en el sistema base** (fuera del entorno de pruebas) apareció como la única solución viable. La decisión se produjo en este caso por descarte de opciones que no cumplen con todos los requisitos buscados.

El empleo de recursos externos al sistema de pruebas creado no es una opción ideal ya que requiere de operaciones de configuración, como la autorización de permisos sobre el servidor X, sobre el sistema que se ejecuta directamente en el hardware y ejerce de anfitrión a los entornos de pruebas.

Existe fragilidad en la solución adoptada, ya que se confía en la salida de comandos del sistema que no garantizan en ningún caso que el formato se mantenga. Se optó, en los casos posibles, por la redundancia de métodos para poder mitigar los posibles efectos de este tipo de cambios y dotar al sistema de mayor robustez.

5.6.1 Restricción importante en la solución

Al necesitar el entorno chroot de un conjunto de software cuyas versiones deben ser exactas o compatibles con el software de aceleración gráfica, que comprende el software controlador de la tarjeta gráfica pero también otras librerías de rendering y herramientas del sistema, la versatilidad del sistema chroot que se puede crear para acceder a la aceleración se reduce considerablemente.

Siendo una versión de una distribución de Linux un conjunto de proyectos software seleccionados a través de unas versiones concretas (ejemplo: Ubuntu Trusty dispone de la versión 3.8 del proyecto Gnome o 7.4 del editor vim) la posibilidad de que en un sistema base de un nodo de Jenkins (que tiene una distribución específica de Ubuntu) disponga de versiones de software de aceleración gráfica compatibles con otras versiones de Ubuntu resulta mínima.

Esta condición de sincronización entre las versiones software del sistema base y el chroot reduce la opción de ejecutar los tests de aceleración gráfica a emplear una distribución de Ubuntu en el entorno de pruebas idéntica a la que emplea el sistema base del nodo. Como ejemplo: un nodo que tiene como sistema base Ubuntu Precise solamente podrá ejecutar los tests de aceleración gráfica de un entorno de pruebas Ubuntu Precise.

La implicación de este problema es la **necesidad de un sistema informático físico por cada distribución soportada por Ubuntu** para poder realizar el testing de distribuciones soportadas al completo.

Capítulo 6

6. **Subproyecto III**: mejoras en el tiempo de creación y el aislamiento del entorno de pruebas

Este capítulo aborda el análisis, diseño y ejecución del tercero de los subproyectos de mejora que tiene relación con la optimización del tiempo empleado por el entorno de pruebas y el grado de aislamiento que tiene el sistema virtual de pruebas respecto al sistema base. Se subdivide en descripción (breve reseña sobre el subproyecto en general), objetivo (metas que se propone el subproyecto), estado inicial (situación en el momento de abordar el subproyecto), soluciones existentes (estado de la cuestión sobre el área tratada en el subproyecto) , solución propuesta (solución elegida y su razonamiento) y conclusiones (corolario final sobre el proceso completo).

6.1 Descripción

Este capítulo aborda el estudio de la mejora sobre el propio entorno virtual inicial (chroot) en dos aspectos fundamentales: por un lado, el tiempo que se emplea en la creación del entorno virtual donde se ejecutan las pruebas automáticas y que resulta crítico para mantener el ciclo de integración continua ágil. Por otro la capacidad de aislamiento del propio entorno virtual respecto al sistema anfitrión con la doble función de mejorar la seguridad del nodo de Jenkins y sobre todo optimizar la aproximación del entorno de pruebas a lo que sería un sistema en real de un usuario final de Gazebo.

6.2 Objetivo

Reducir el tiempo que el sistema emplea en preparar el entorno de pruebas, como objetivo principal, y mejorar el grado de aislamiento respecto al sistema base instalado en el nodo de Jenkins encargado de crear los entornos de pruebas automáticas, como objetivo secundario.

6.3 Estado inicial

En la situación inicial, el entorno de pruebas dentro del sistema Linux es creado mediante el empleo de un chroot por el sistema base del nodo de Jenkins que ejecuta el trabajo de testing.

El entorno de pruebas debe disponer de todo lo necesario para la correcta compilación del software y la ejecución de las pruebas automáticas, así como otras herramientas empleadas en el proceso de integración continua como pueden ser un analizador de código estático. Los pasos que abarca el proceso de creación del entorno de pruebas son:

1. Sistema base: creación del chroot para el sistema Ubuntu Linux utilizado en el testing o actualización del mismo si existía
2. Entorno pruebas chroot: instalación de repositorios de paquetes .deb adicionales
3. Entorno pruebas chroot: instalación de dependencias de Gazebo utilizando paquetes binarios (.deb)
4. Entorno pruebas chroot: configuración mediante cmake
5. Entorno pruebas chroot: compilación del código C++ de Gazebo
6. Entorno pruebas chroot: ejecución de pruebas
7. Entorno pruebas chroot: ejecución de cppcheck
8. Sistema base: exportar test y resultados al lugar correcto para ser leídos por Jenkins

Se podría considerar que la creación del entorno de pruebas abarca desde el punto 1 en el que sistema base crea el 'sistema virtual' hasta el punto 4 en que se termina la configuración y el sistema está listo para comenzar con la compilación.

6.4 Soluciones existentes

El objetivo fundamental de creación de un entorno de pruebas es la recreación aproximada de un sistema en producción (en este caso, un sistema de usuario del simulador Gazebo). Para conseguirlo, los procesos de integración continua tradicionalmente han empleado algún tipo de virtualización que permita la creación un sistema desde ceros. Esto tiene la doble función de proporcionar **repetibilidad** y aproximar el sistema a lo que sería un sistema en producción, sin alteraciones previas relativas a la instalación o ejecución del simulador.

La virtualización es un concepto amplio que ha existido en la tecnología de sistemas operativos desde los primeros inicios en los años sesenta [16]. Dentro del término virtualización se pueden encontrar diferentes variantes dependiendo del grado que proporcionan de emulación hardware. Este documento realiza un completo detalle de estas tecnologías en la sección 3.3 (Entornos virtuales)

Partiendo de la existencia inicial de un entorno creado por un chroot y de los requisitos que este subproyecto fija como objetivos (la reducción del tiempo de creación del entorno de pruebas y su aislamiento respecto del sistema base) se realiza un análisis de qué opciones podrían cumplir con ambos puntos (ver tabla 3).

Sistema de virtualización	Impacto en el rendimiento	Grado de Aislamiento
chroot (inicial) *virtualización a nivel de SO*	Muy Bajo, similar a ejecutar instrucciones en un sistema no virtual.	Bajo: Solamente efectivo a nivel de sistema de ficheros. No hay limitación de otros recursos compartidos.
VirtualBox *Para-virtualización albergada*	Impacto muy alto sobre rendimiento [18].	Elevado grado de aislamiento: procesos propios, hardware emulado
KVM *Para-virtualización nativa*	Impacto alto sobre rendimiento, menor que VirtualBox [18].	Elevado: procesos propios, hardware emulado, etc.
LXC *virtualización a nivel de SO*	Muy Bajo, similar a ejecutar instrucciones en un sistema no virtual.	Medio: aislamiento de recursos pero no hardware emulado
Docker *virtualización a nivel de SO*	Muy Bajo, similar a ejecutar instrucciones en un sistema no virtual.	Medio: aislamiento de recursos pero no hardware emulado

Tabla 3: comparativa entre sistemas virtualizados de impacto sobre rendimiento y grado de aislamiento

Al ser un chroot una técnica que no proporciona ningún tipo de emulación hardware real sino que simplemente redirecciona la raíz del sistema a un directorio determinado, teóricamente nos encontramos con el problema de que el empleo de cualquiera de los sistema que sí ejecutan emulación en sus diferentes niveles (virtualización, para-virtualización, etc.) van a resultar más lentos que el propio chroot.

Por otro lado, todas las soluciones alternativas proporcionan un grado de aislamiento respecto del sistema base mayor que el dispuesto por un chroot. Las distintas opciones ofrecen un abanico amplio de aislamiento, desde la segregación de recursos y procesos hasta la emulación de distintas partes de hardware.

6.5 Solución propuesta

Para alcanzar una solución satisfactoria en primer lugar cabe analizar el impacto sobre el rendimiento que figuraba como el **objetivo principal para minimizar tiempos en el ciclo de integración continua**.

Cualquier solución que impacte de manera considerable en el tiempo de creación del entorno de pruebas debería ser descartada de acuerdo a los objetivos iniciales. El objetivo secundario es mejorar el grado de aislamiento del entorno respecto del sistema base con la intención principal de aproximar el entorno de pruebas a un sistema de usuario final sin que ningún aspecto del sistema base o su configuración pudiera interferir en las pruebas.

La solución propuesta ha sido impulsada por la necesidad de reducir el tiempo de creación del entorno de pruebas como objetivo principal. Con este requisito presente, las opciones de reemplazar el chroot por un entorno de virtualización o paravirtualización son desplazadas como opciones prioritarias desde la asunción que cualquier instrucción nativa es necesariamente más rápida de ejecución que si se virtualiza desde algunas de las técnicas existentes.

Esta argumentación sitúa como **opciones prioritarias aquellas que implementan una virtualización a nivel de sistema operativo**, que teóricamente son las variantes que producen un incremento en el tiempo de creación del entorno de pruebas menor. En este apartado existen dos soluciones técnicas de entre las valoradas que implementan esta virtualización a nivel de sistema operativo: LXC y Docker.

Si se intenta comparar ambas opciones directamente, LXC y Docker, surge el problema de que una de ellas, Docker, es realmente un derivado y empleó a la otra, LXC. Docker en sus implementaciones iniciales empleaba LXC directamente y añadía funcionalidades adicionales sobre la capacidad de LXC [19]. Si se amplía esta línea de razonamiento, también se podría entender a LXC como una desarrollo reciente de la propia tecnología chroot, extendiendo las capacidades de la misma. Con esta idea quasi-lineal de desarrollo de un mismo paradigma de virtualización a nivel de sistema operativo, chroot -> LXC -> Docker, la propuesta de solución pasa por emplear la última de las tecnologías y aprovechar al máxima el desarrollo que se ha producido en el área: Docker.

6.5.1 Adaptación a docker

Partiendo de la implementación existente de entorno de pruebas basado en el chroot (ver anexo 2) se diseña una transición desde el mismo al sistema Docker (ver figura 19).

Uno de los fundamentos de docker para la creación de sistemas virtuales es la codificación de las acciones de creación en un fichero llamado Dockerfile, que será procesado por docker mediante su comando build generando una imágen que no es más que un sistema personalizado listo para ser empleado.

El otro paso fundamental es la propia ejecución de aplicaciones dentro de las imágenes creadas o disponibles, mediante el empleo del comando *run* de docker. El proyecto Gazebo tiene particularidades que requerirán de personalizar ambas fases.

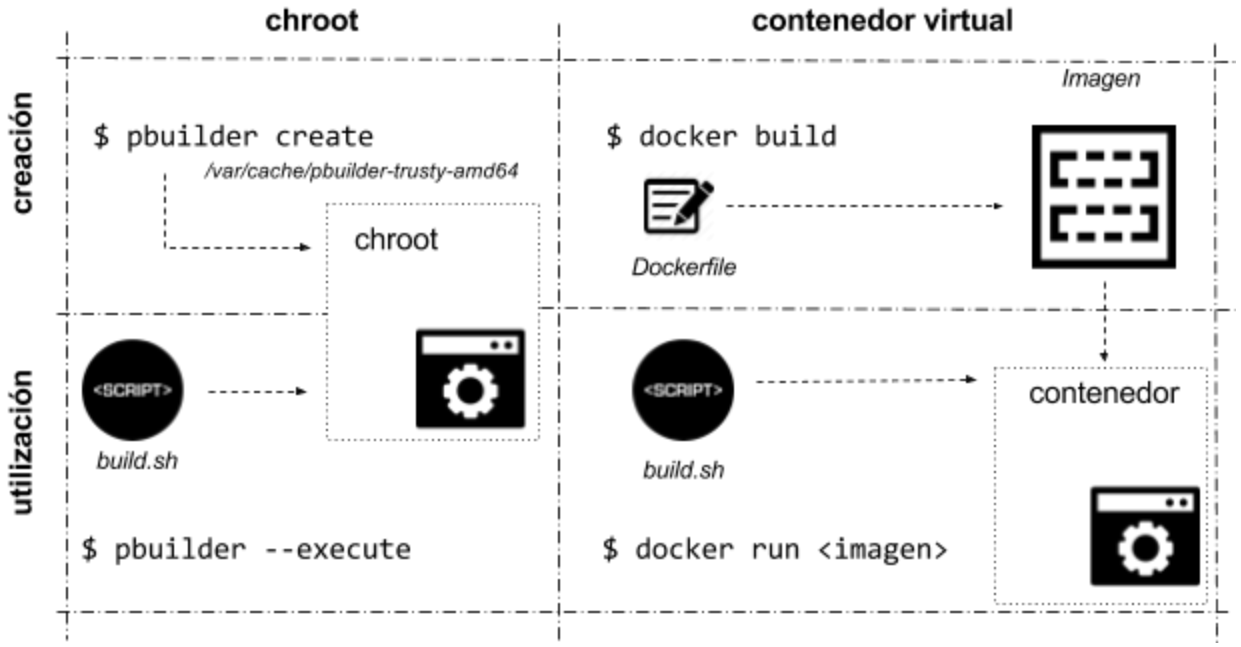


Figura 19: comparación de los pasos para la creación y empleo del entorno virtual en chroot y docker

6.5.2 Creación de la imágen docker

Para imitar las copias de sistemas operativos que manejaba chroot mediante la herramienta pbuilder, en docker se dispone del concepto de imágenes predefinidas que pueden ser utilizadas directamente como sistema base sobre el que interactuar. En el caso inicial de Gazebo, docker dispone en su Hub (repositorio público de imágenes docker) de soporte oficial de todas las versiones de Ubuntu disponibles que se pueden utilizar directamente o en composición de imágenes derivadas mediante empleando la directiva *FROM* que fija la imágen de partida.

Durante la implementación aparecieron algunas de las limitaciones de docker. En concreto la falta de soporte nativo para la arquitectura x86 de 32 bits. Para solucionar el problema se emplearon imágenes generadas por el proyecto ROS disponibles a través del Hub de la OSRF en Docker que integra Qemu (empleando los binarios estáticos de Qemu) con Docker y permite disponer de arquitecturas alternativas como son i386 o ARM.

El código bash que genera la parte inicial del Dockerfile, implementando la selección apropiada de la imagen base, es el siguiente:

```
# Select the docker container depending on the ARCH
case ${ARCH} in
  'amd64')
       FROM_VALUE=${LINUX_DISTRO}:${DISTRO}
       ;;
  'i386')
  'armhf' | 'arm64' )
       # Use OSRF Hub for integration of qemu
       FROM_VALUE=osrf/${LINUX_DISTRO}_${ARCH}:${DISTRO}
       ;;
  *)
       echo "Arch unknown"
       exit 1
esac

cat > Dockerfile << DELIM_DOCKER
# Docker file to run build.sh

FROM ${FROM_VALUE}
MAINTAINER Jose Luis Rivero <jose.luis.rivero.partida@gmail.com>

# setup environment
ENV LANG C
ENV LC_ALL C
ENV DEBIAN_FRONTEND noninteractive
ENV DEBFULLNAME "OSRF Jenkins"
ENV DEBEMAIL "jose.luis.rivero.partida@gmail.com"
DELIM_DOCKER
```

6.5.3 Transformación del script de compilación y pruebas

El sistema inicial disponía también de un script llamado *build.sh* encargado de instalar las dependencias de Gazebo, configurar el código (cmake), compilar el código y ejecutar las pruebas automáticas.

En docker se puede conservar la mayor parte del script ya que realmente solo el primero de los puntos (instalación de dependencias) interpela al entorno virtual de pruebas. El resto de pasos (configuración, compilación, pruebas) tienen relación con las ejecuciones del ciclo de integración continua y por tanto quedan fuera del marco de reforma del entorno virtual de pruebas.

Eliminando la instalación de dependencias (que se trasladará a la creación de Dockerfile) la creación de build.sh se implementa así:

```
cat > build.sh << DELIM
# Normal cmake routine for Gazebo
rm -rf $WORKSPACE/build $WORKSPACE/install
mkdir -p $WORKSPACE/build $WORKSPACE/install
cd $WORKSPACE/build
cmake
-DPKG_CONFIG_PATH=/opt/ros/fuerte/lib/pkgconfig:/opt/ros/fuerte/stacks/visualization_common/og
re/ogre/lib/pkgconfig -DCMAKE_INSTALL_PREFIX=$WORKSPACE/install $WORKSPACE/gazebo
make -j
make install
. $WORKSPACE/install/share/gazebo-1.*/setup.sh
LD_LIBRARY_PATH=/opt/ros/fuerte/lib:/opt/ros/fuerte/stacks/visualization_common/ogre/ogre/lib
make test ARGS="-VV" || true

# Step 3: code check
cd $WORKSPACE/gazebo
sh tools/code_check.sh -xmldir $WORKSPACE/build/cppcheck_results || true
DELIM
```

Una vez generado el script es necesario copiar al entorno virtual (directiva *COPY* de docker) y dotarlo de permisos de ejecución para poder invocarlo al emplear el comando *docker run*:

```
cat >> Dockerfile << DELIM_DOCKER
COPY build.sh build.sh
RUN chmod +x build.sh
DELIM_DOCKER
```

6.5.4 Dependencias y la caché de docker

La instalación de dependencias que en el punto previo fue eliminada del script build.sh necesita ser integrada dentro del archivo Dockerfile que construye el sistema virtual base para pruebas. Es importante remarcar que cada una de las instrucciones contenidas en Dockerfile hará que Docker guarde la imagen parcial en su caché para poder ser recuperada en presencia de un fichero Dockerfile con las mismas instrucciones.

En los scripts empleados para generar la imagen de docker se definen variables de entorno que almacenan los paquetes de dependencias. *BASE_DEPENDENCIES* es la variable que define los paquetes básicos para la ejecución de integración sobre cualquier software, *GAZEBO_BASE_DEPENDENCIES* (reducida en el código a continuación para mantener una extensión razonable) contiene los valores necesarios para la compilación de Gazebo. *PACKAGES_CACHE_AND_CHECK_UPDATES* es la variable que contiene la lista de paquetes (básicamente la unión de las dos variables anteriores más los paquetes necesarios para replicar el soporte de aceleración gráfica si es necesario) que serán procesados por docker como parte de Dockerfile y por lo tanto almacenados en la memoria caché:

```
BASE_DEPENDENCIES="build-essential \
                       cmake          \
                       debhelper      \
                       cppcheck       \
                       xsltproc       \
                       python         \
                       gnupg2         \
                       python-empy    \
                       python-argparse \
                       debhelper"

GAZEBO_BASE_DEPENDENCIES="   libfreeimage-dev                    \
                             libprotoc-dev               \
                             libprotobuf-dev             \
                             protobuf-compiler           \
                             …"
# Packages that will be installed and cached by docker. In a non-cache
# run below, the docker script will check for the latest updates
PACKAGES_CACHE_AND_CHECK_UPDATES="${BASE_DEPENDENCIES} ${DEPENDENCY_PKGS}"

if $USE_GPU_DOCKER; then
  PACKAGES_CACHE_AND_CHECK_UPDATES="${PACKAGES_CACHE_AND_CHECK_UPDATES}
                                    ${GRAPHIC_CARD_PKG}"
fi
```

La gestión que hace Docker de su memoria caché es muy útil para no tener que instalar las dependencias de Gazebo en cada ejecución del ciclo de integración continua. Se instalarán en la primera ejecución invocada por *docker build* y, si la caché no se borra hasta la siguiente ejecución, **podrá reutilizarse la imagen binaria que ya contiene las dependencias**, con el consiguiente ahorro en tiempo de descarga de paquetes e instalación de los mismos.

Esta caché binaria que almacena la imagen preprocesada presenta alguna limitación importante que es necesario solucionar. Al ser una **memoria caché el reto de mayor relevancia es el mantener actualizada la misma** cuando aparecen cambios en el contenido que ha sido almacenado. Si se instalara la lista de paquetes definidos anteriormente como dependencias (mediante la directiva *RUN* de docker) sin ninguna otra medida, se dispondría de un sistema de pruebas diferente a la realidad tan pronto como alguna de las dependencias sea actualizada en Ubuntu o alguna nueva versión del resto de software de la OSRF sea liberada. Las consecuencias serían una divergencia creciente respecto a los sistemas de usuario, algo no tolerable en la solución final y que figuraba como objetivo secundario (que no opcional).

Para solucionar el problema de disponer de memorias caché desactualizadas pero seguir disfrutando de los beneficios del empleo de la misma se implementaron dos medias: por un lado, la caché se actualizaría completamente cada 30 días desde la generación de la anterior. Por otro lado, después de realizar la primera instalación (o disponer de la imagen binaria de la caché) para **asegurar la total actualización de cada ejecución** se implementa una segunda instalación de las dependencias después de realizar un paso de invalidación de la caché, lo cual siempre ejecutará la instrucción y no permitirá el empleo de imágenes guardadas. La expectativa de nuevos paquetes en esta segunda instalación es muy baja con lo cual la ventaja de disponer de la caché en la primera instalación sigue vigente.

Como nota de implementación, para invalidar la caché en docker basta con disponer de una directiva diferente en el fichero Dockerfile. Para conseguir esto se emplea el comando echo y una variable que sea distintiva por cada mes (*MONTH_YEAR_STR*) y para la invalidación total el mismo comando echo con un valor aleatorio que genere suficiente entropía (comando bash *RANDOM*). La implementación de esta idea en el Dockerfile sería:

```
cat >> Dockerfile << DELIM_DOCKER
# Invalidate cache monthly
# This is the first big installation of packages on top of the raw image.
# The expectation of updates is low and anyway it is cached by the next
# update command below
RUN echo "${MONTH_YEAR_STR}" \
 && (apt-get update || (rm -rf /var/lib/apt/lists/* && apt-get update)) \
 && apt-get install -y ${PACKAGES_CACHE_AND_CHECK_UPDATES} \
 && apt-get clean


# This is killing the cache so we get the most recent packages if there
# was any update. Note that we don't remove the apt/lists file here since
# it will make to run apt-get update again
RUN echo "Invalidating cache $(( ( RANDOM % 100000 )  + 1 ))" \
 && (apt-get update || (rm -rf /var/lib/apt/lists/* && apt-get update)) \
 && apt-get install -y ${PACKAGES_CACHE_AND_CHECK_UPDATES} \
 && apt-get clean
DELIM_DOCKER
```

El código completo de la implementación docker de la versión 2 de Gazebo está disponible en el anexo 3.

Una vez creado el Dockerfile, es necesario invocar el comando build para construir la imagen. Cada imagen tendrá una etiqueta univoca que la identifica, el sistema implementado emplea la variable DOCKER_TAG para identificar cada imagen. El comando build es sencillo, necesita ejecutarse en el directorio que contiene el fichero Dockerfile:

```
sudo docker build --tag ${DOCKER_TAG} .
```

6.5.5 Ejecución de un contenedor docker

La ejecución de la imagen creada anteriormente identificada por el contenido de la variable DOCKER_TAG no reviste gran dificultad. Únicamente cabe reseñar que para poder acceder satisfactoriamente a la GPU se emplea la opción *--priviledge*, además se mapean tanto */dev/log* como */run/lo*g en modo de lectura para que la instalación y ejecución del sistema de log dentro del contenedor no resulte en error. Existen también alguna variables de entorno útiles dentro del contenedor, como son el directorio de trabajo de jenkins (*WORKSPACE*) y el acceso al mismo, o la definición del tipo de terminal y la inclusión de un pseudoterminal (*--tty*) para interactuar con el sistema. También se incluye la opción de eliminar automáticamente el contenedor cuando termina la ejecución (*--rm*).

Para finalizar, se indica la ejecución del fichero build.sh a través del intérprete bash. El código que implementa estas ideas es:

```
sudo docker run --priviledge  \
               -e WORKSPACE=${WORKSPACE} \
               -e TERM=xterm-256color \
               -v ${WORKSPACE}:${WORKSPACE} \
               -v /dev/log:/dev/log:ro \
               -v /run/log:/run/log:ro \
               --tty \
               --rm \
               ${DOCKER_TAG} \
               /bin/bash build.sh
```

6.6 Conclusiones

Para estudiar el resultado de la nueva implementación se analizan por separado los objetivos descritos en la sección inicial del capítulo: objetivo principal de aceleración del bucle de integración continua y el objetivo secundario de mejora en la aproximación del entorno virtual de pruebas a los sistemas de usuario final.

Para el objetivo principal se puede analizar la reducción de tiempo desde una perspectiva puramente teórica o emplear una aproximación pragmática y obtener tiempos reales de ejecución. Se desarrollan ambas.

Empleando una aproximación puramente teórica el hecho de disponer de imágenes binarias de sistemas Ubuntu en lugar de tener que construirlas instalando los paquetes uno por uno podría suponer una ventaja si consideramos el rendimiento de red estable. Se ahorran las operaciones típicas de procesos de instalación: entrada/salida a disco, descompresión de ficheros, etc. También desde un punto teórico, que la implementación en docker permite el empleo de la caché supone un ahorro de operaciones sobre el sistema virtual de pruebas. Especialmente remarcable es el hecho de haber **desplazado la instalación de dependencias desde el script build.sh hacia el fichero Dockerfile** ya que hace que la descarga e instalación de todos los paquetes pase a realizarse en una única ejecución (renovada cada mes) en lugar de hacerse en cada ejecución del entorno de pruebas automáticas. Asumiendo válido este razonamiento, la aproximación teórica muestra mejora sobre el tiempo inicial pero no se puede conocer la magnitud del mismo.

Para conocer la magnitud de la ganancia se han realizado una serie de experimentos de instalaciones de Gazebo5 empleando los entornos chroot y docker disponibles en los anexos 2 y 3.

El entorno de ejecución de las pruebas fue un portátil ASUS UX51VZ - Core i7 3612QM / 2.1 GHz - Ubuntu Linux 16.04t - 8 GB RAM - NVIDIA GeForce GT 650M.

Se comparan tiempos totales de cada ejecución del ciclo de integración continua. Se muestra en los gráfico el tiempo total empleando por una ejecución del ciclo de integración al completo. Cuando es necesaria la creación del entorno virtual de pruebas (*build*), como cuando el entorno de pruebas ya existe y se procede a nuevas ejecuciones sobre él (*update*).

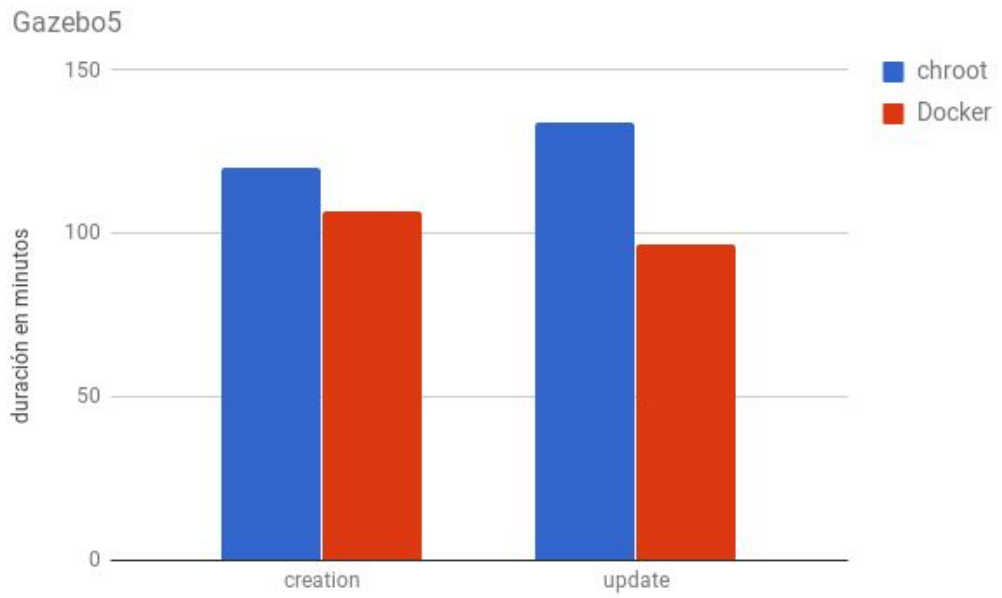


Figura 19 (bis): gráfico de comparación de tiempos entre chroot y docker

Se observa una reducción total del tiempo empleado en la **creación del entorno y la ejecución de las pruebas automáticas** de 13.28 minutos de media al emplear la solución de docker en contrate con la inicial de chroot. Esto supone una **reducción del 12.44% del tiempo** empleado.

El segundo de los objetivos definidos por este capítulo consistía en la mejora del propio entorno de pruebas, dotándola de mayor aislamiento sobre el sistema base para aproximar el entorno de pruebas a un sistema de usuario final en el que no se ha intervenido desde el proyecto Gazebo en manera previa alguna.

La solución propuesta ha continuado la línea de emplear tecnologías virtuales con un grado de impacto en el rendimiento bajo, concretamente la virtualización de a nivel de sistema operativo. La sustitución de un entorno chroot, una de las primeras tecnología que implementó en el núcleo de Linux este tipo virtualización por Docker, es una de las últimas tecnologías que implementa la misma aproximación y está en pleno auge en este año 2017. La sustitución de

una por otro ha permitido aislar recursos que al inicio eran completamente accesibles por el entorno chroot: procesos del sistema base, interacción con los distintos demonios activos, etc.

Capítulo 7

7. **Subproyecto IV**: automatizar la creación y mantenimiento de los trabajos de Jenkins

Este capítulo aborda el análisis, diseño y ejecución del último de los subproyectos de mejora que tiene relación con la mejora de la gestión de las configuraciones que tiene cada trabajo en el servidor de integración continua, pasando de un modelo manual o configuraciones generadas automáticamente. Se subdivide en descripción (breve reseña sobre el subproyecto en general), objetivo (metas que se propone el subproyecto), estado inicial (situación en el momento de abordar el subproyecto), soluciones existentes (estado de la cuestión sobre el área tratada en el subproyecto) , solución propuesta (solución elegida y su razonamiento) y conclusiones (corolario final sobre el proceso completo).

7.1 Descripción

Las medidas expuestas en este subproyecto tratan de mejorar la situación generada por el crecimiento en el número de trabajos de Jenkins debido a la ampliación del número de plataformas de testing y la separación de partes del simulador que pasan a ser librerías propias, requiriendo de sus propios trabajos en Jenkins para implementar su ciclo de integración continua.

7.2 Objetivo

Facilitar la creación y actualización de trabajos de Jenkins en las diferentes plataformas existentes y aquellas que serán liberadas en el futuro, sin necesidad de realizar procesos manuales con la problemática e inconvenientes que implican.

7.3 Estado inicial

En el capítulo 3 (Estado de la cuestión) se detalla como en Jenkins la manera standard de crear o editar trabajos se ejecuta mediante la utilización de la interfaz web y configurando manualmente los mismos, empleando las herramientas de formulario web de las que dispone la interfaz.

En instalaciones en las que el número de trabajos es limitado (hasta varias docenas) resulta un método de gran usabilidad y con un coste en la curva de aprendizaje muy bajo. Como contrapartida, en instalaciones con un número de trabajos medio (entendido como por encima de la cincuentena) o alto (desde varios centenares en adelante) el mantenimiento del conjunto de trabajos, la creación de nuevos o la eliminación de existentes resultan muy costosos en tiempo.

Hay que destacar que además del propio Gazebo, la Open Source Robotics Foundation desarrollaba en paralelo SDFormat (ver referencia en el apartado 2.1 Estado de la cuestión) desde el inicio de este proyecto.

Analizando el caso de evolución de Gazebo: inicialmente en el año 2012 coincidían en el tiempo dos versiones de Ubuntu que Gazebo soportaba oficialmente: Precise (12.04) y Quantal (12.10) tanto para 64 bits como para 32 bits. Esto reducía el espectro de opciones a soportar por el servidor de testing a menos de una decena.

Este escenario inicial se vió alterado por distintos motivos:

- En primer lugar existieron y existen todavía ciclos en los que **Ubuntu soporta oficialmente hasta cuatro versiones al mismo tiempo** (ver figura 20) . Gazebo en su política de distribución soporta y dispone oficialmente de paquetes .deb para todas las versiones en activo de Ubuntu. Estos ciclos disparan exponencialmente el número de trabajos a soportar en determinados períodos de tiempo.

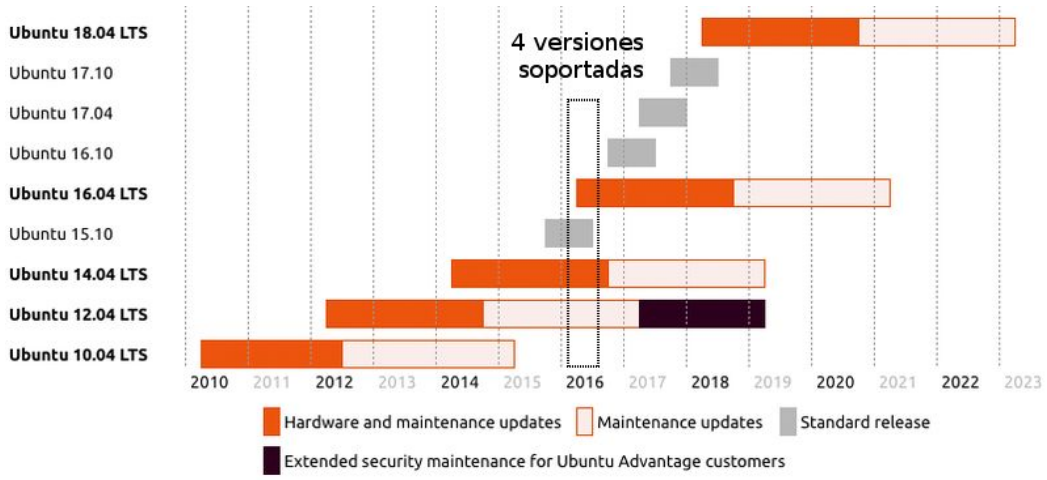


Figura 20: versiones de Ubuntu planificadas y anotación sobre versiones soportadas en paralelo

- A finales de 2013 el proyecto Gazebo adopta el esquema de versiones conocido *"Semantic Versioning"* (versionado semántico) creado por Tom Preston-Werner y libera su versión 2.0. Desde ese momento **varias versiones de Gazebo son oficialmente soportadas al mismo tiempo**, haciendo crecer el número de trabajos en el servidor de integración continua.
- En 2016 el equipo de desarrollo del simulador Gazebo emprende una **refactorización que lleva a la creación de nuevos proyectos software (librerías ignition-robotics)** partiendo del código que originalmente formaba parte del proyecto Gazebo. Esto hace que la granja de pruebas amplíe el número de proyectos bajo testing pasando de simplemente dos (Gazebo y SDFormat) a un mínimo de cuatro (con ignition-math e ignition-transport).
- Las arquitecturas soportadas inicialmente eran las basadas en x86 tanto para 32 como para 64 bits. Con el paso de los años el crecimiento en el empleo de las arquitecturas basadas en procesadores ARM ha sido enorme. **Estas arquitecturas basadas en ARM pasaron a ser soportadas por Gazebo**.

Estos factores comenzaron a dificultar el mantenimiento de trabajos activos en diversos aspectos: el tiempo a invertir en cada ocasión en que una nueva plataforma era soportada (por ejemplo con una nueva versión de Ubuntu) crecía exponencialmente (ver tabla 4). También resultaba problemático detectar todos los trabajos que se deberían eliminar o deshabilitar cuando el soporte para alguna de las plataformas o versiones de Gazebo quedaba obsoleta.

Trabajos registrados inicialmente			Trabajos registrados 2014		
Software	***Versión***	***Plataformas***	***Software***	***Versión***	***Plataformas***
Gazebo	1.8	Ubuntu Precise and Ubuntu Quantal	**Gazebo**	2.0 3.0	2.0: Ubuntu Precise, Quantal 3.0: Ubuntu Precise, Raring, Saucy
SDFormat	1.4		**SFormat**	1.4 2.0	1.4: Ubuntu Precise, Quantal 2.0 Ubuntu Precise, Raring, Saucy
			Ignition math	1.0	Ubuntu Precise, Raring, Saucy

Tabla 4: comparativa de soporte de software en la granja de automatización entre el estado inicial en 2013 y mediados del año 2014.

7.3 Soluciones existentes

La búsqueda de soluciones a la intervención manual vía interfaz web paso por sustituir esta interacción por algún método que pudiera realizar cambios en bloque sobre un conjunto de trabajos. Existen diferentes alternativas para poder llevar a cabo este tipo de operaciones secuenciales:

7.3.1 Editar directamente los ficheros de configuración

Al almacenar el servidor Jenkins la configuración de cada uno de los trabajos en un archivo XML algunas alternativas pasarían por gestionar cambios colectivos desde el propio sistema de ficheros **editando directamente el conjunto de ficheros config.xml** almacenados en el disco duro del sistema que alberga el servidor.

Simplemente empleando herramientas básicas de un sistema Linux (como puedan ser *find* y *sed*) se pueden realizar acciones sobre los ficheros de configuración. Teniendo en cuenta que la disposición en el sistema de ficheros tiene un directorio jobs/ dentro del propio direction principal de Jenkins (típicamente */var/lib/jenkins*) y dentro de este directorio *jobs/* existen un subdirectorio por cada trabajo registrado en el servidor, con el nombre del propio trabajo (ej: *jobs/gazebo-default-devel-precise*). Si se ha seguido una nomenclatura consistente, eliminar todos los trabajos que se ejecutan sobre la distribución Ubuntu Precise supondría ejecutar un par de comandos sencillos:

```
cd  $JENKINS_HOME/jobs/
rm *-*-*-precise
```

Para que el servidor Jenkins tenga en cuenta las modificaciones sobre los ficheros de configuración hay que ejecutar la opción “Reload Configuration From Disk” que se encuentra en el menú principal, dentro de la opción “Manage Jenkins” (ver figura 21).

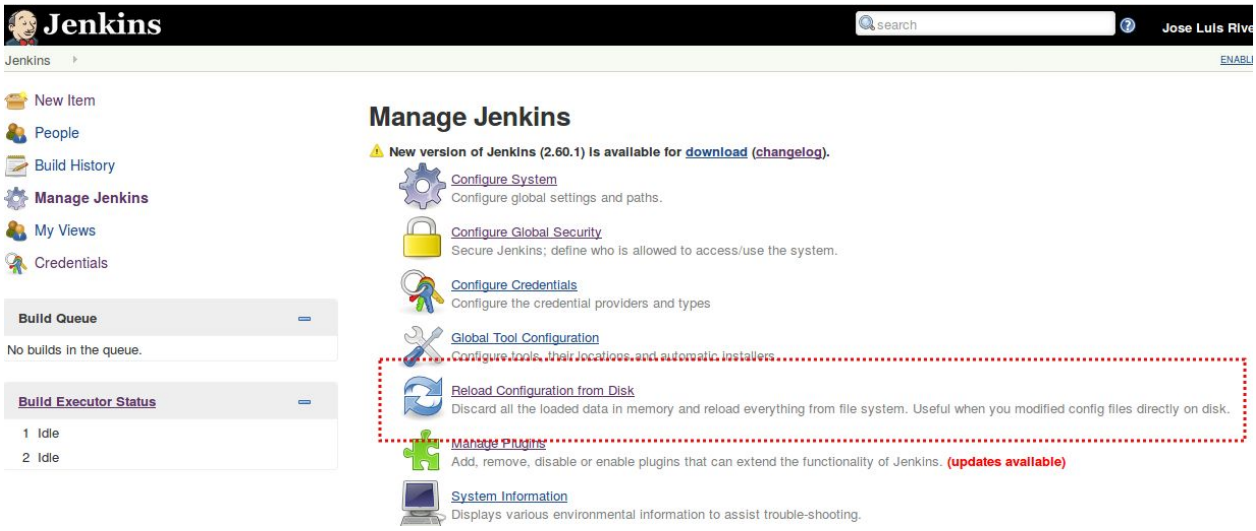


Figura 21: captura de pantalla de la sección “Manage Jenkins” y la opción “Reload Configuration from Disk”

Algunas operaciones de edición directas sobre los propios ficheros de configuración requerirían de un comando algo más complejo. Por ejemplo reemplazar todos los trabajos que se ejecutan sobre Ubuntu Quantal por Ubuntu Trusty supondría:

```
# 1. editar sobre los trabajos quantal el reemplazo del nombre y otras configuraciones que pudieran incluir el nombre de la distribución
find *-*-*-quantal -name config.xml -exec sed -i -e ‘s/quantal/trusty/g’ {} \;
# 2. modificar el nombre del subdirectorio
for d in $(find . -name '*-quantal'); do
  mv $d ${d/quantal/trusty}
done
```

Y nuevamente volver a cargar la configuración desde la interfaz de Jenkins.

7.3.2 Empleo de plantillas para generar los ficheros de configuración

Si mantenemos la idea de incidir directamente sobre los ficheros de configuración, una opción sería generar, mediante el empleo de código programable, todos los ficheros de configuración al completo, ignorando totalmente las operaciones web que proporciona Jenkins.

Este tipo de prácticas en las que se automatiza la configuración de un servidor se conocen en el ámbito DevOps (*development operations*) bajo el nombre de “Infraestructura como código” [20] [(infrastructure as code) o “Configuración como código (configuration as code) y nos acercan a una realidad dinámica de operaciones sobre la infraestructura.

Uno de los proyecto que alberga la Open Source Robotics Foundation, el framework conocido como ROS (Robot Operating System), implementa para su granja de pruebas (compuesta por miles de trabajos) siguiendo este modelo basado en no generar ningún tipo de configuración utilizando la interfaz web y, en lugar de esto, emplear un sistema de programación para generar y mantener todos sus trabajos. El equipo de ROS utiliza para toda la implementación de sus trabajos en Jenkins una software de desarrollo propio, basado en python, que es capaz de generar todas las configuraciones e interactuar directamente con Jenkins para cargarlas en el servidor y modificar todos los aspectos necesarios para que la nueva configuración entre en efecto.

Este software (utiliza el nombre de *ros_buildfarm*) es un conjunto de scripts en python que emplean una librería auxiliar de plantillas (llamada empy [21]) para generar dinámicamente la configuración XML de cada trabajo de Jenkins (ver figura 22), empleando las propias variables de python.

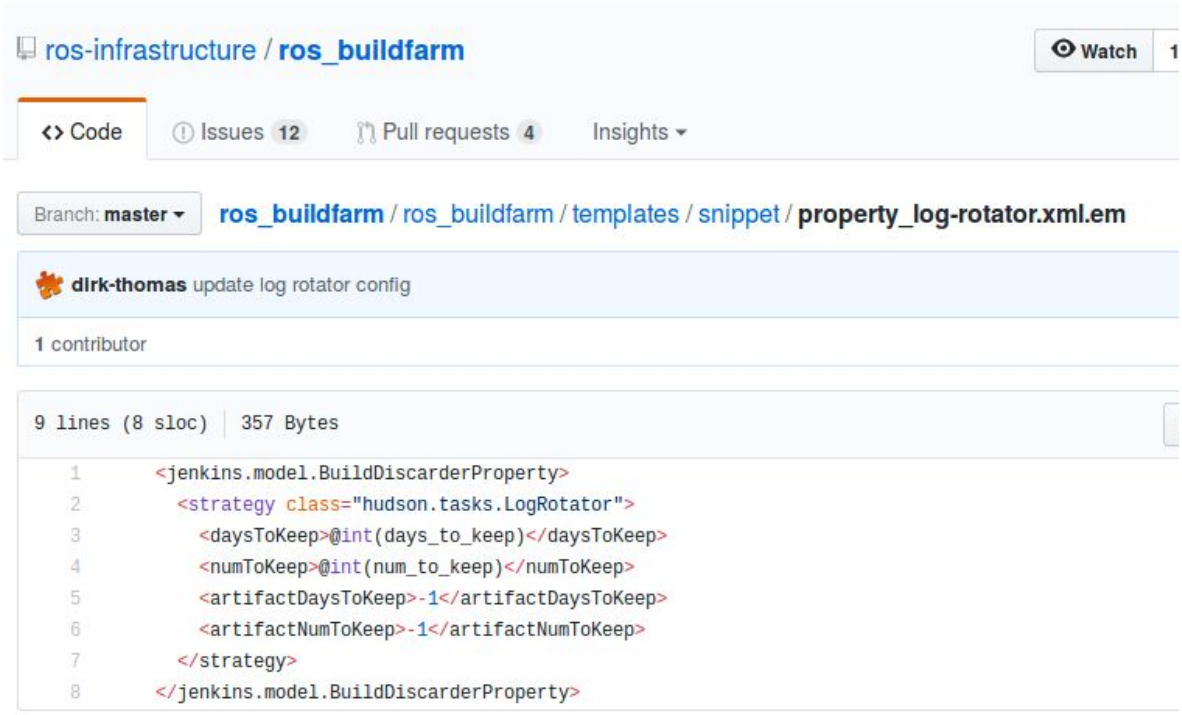


Figura 22: repositorio del proyecto ros_buildfarm y código de plantilla en empy

7.3.3 Solución nativa: Jenkins DSL

Jenkins es un proyecto maduro empleado a nivel mundial por miles de diferentes organizaciones. Algunas de ellas son grandes multinacionales con centenares de servidores y una organización informática mucho mayor que el pequeño proyecto de mejoras sobre el simulador Gazebo que se detalla en este documento. Una de las vías para encontrar **soluciones a problemas recurrentes es el análisis de cómo se han intentado solucionar previamente**.

La conocida empresa de entretenimiento Netflix, dedicada principalmente a las series y videos en streaming desde Internet, dispone de una de las instalaciones de Jenkins más grandes que se conocen (hasta de 25 masters diferentes) y los desarrolladores de la misma han proporcionado abundante información sobre cómo gestionan la instalación desde su blog en medium [22].

Para solucionar el problema de escalabilidad y mantenibilidad de los trabajos, Netflix emplea una solución nativa de que Jenkins desarrolló desde 2012 conocida como **Jenkins DSL** (Domain Specific Language).

Jenkins DSL está compuesto por dos partes: por una lado la propia definición de un lenguaje creado exclusivamente para generar configuraciones de Jenkins (de ahí la denominación como DSL) y por otro lado un plugin que procesa los scripts de este lenguaje y genera las configuraciones y trabajos especificados. El lenguaje está basado en Apache Groovy, que a su vez es un lenguaje pensado para interactuar con Java (el código del servidor Jenkins es Java).

El objetivo de Jenkins DSL es el de generar programáticamente y de manera sencilla la configuración de los trabajos y su mantenimiento, permitiendo compartir numerosas secciones y bloques de configuración entre los distintos trabajos. La idea es que con pocas líneas de código muy descriptivo se puedan generar configuraciones de trabajos en breves minutos (ver ejemplo en figura .

```
def gitUrl = 'git://github.com/jenkinsci/job-dsl-plugin.git'

job('PROJ-unit-tests') {
    scm {
        git(gitUrl)
    }
    triggers {
        scm('*/15 * * * *')
    }
    steps {
        maven('-e clean test')
    }
}

job('PROJ-sonar') {
    scm {
        git(gitUrl)
    }
    triggers {
        cron('15 13 * * *')
    }
    steps {
        maven('sonar:sonar')
    }
}
```

Figura 23: ejemplo proporcionado por el proyecto Jenkins DSL que muestra código propio para la generación de dos trabajos

Durante los años 2016 y 2017, el proyecto Jenkins DSL ha tenido un desarrollo muy activo y gran parte de los plugins no incluídos en el núcleo de Jenkins están también soportados y pueden ser empleados como un elemento más del propio lenguaje. Si tenemos en cuenta que los plugins recogen gran parte de funcionalidades importantes, este hecho es especialmente relevante.

Para todos aquellos plugins o configuraciones no soportados oficialmente por Jenkins DSL como parte del lenguaje, existe la posibilidad de “inyectar” el código necesario para que directamente aparezca en la configuración XML final y de esta manera no tener que dependender de herramientas extra u otro tipo de procesos fuera del entorno del propio Jenkins DSL.

7.4 Solución propuesta

Analizando las soluciones propuestas por el apartado anterior, es sencillo encontrar un buen número de argumentos en contra de adoptar como solución definitiva y duradera las operaciones realizadas directamente sobre el sistema de ficheros empleando utilidades linux como *sed*, *find* o comandos nativos de *bash*, como detalla la primera de las propuestas.

En primer lugar, las operaciones de modificación o creación de configuraciones en Jenkins resultarían complejas de realizar empleando ediciones sobre el fichero de texto con las utilidades de línea de comando UNIX antes mencionadas. Estas utilidades carecen de la lógica suficiente para respetar la integridad del formato XML, se requeriría de alguna herramienta adicional que proporcionara la capacidad de conocer si el formato se respetó. El empleo de expresiones regulares para realizar algunas de las ediciones es frágil, ya que los valores codificados en el XML pueden perfectamente variar con el tiempo sin aviso previo.

Descartando esta primera propuesta por las inconsistencias que implicaría llevarla más allá de la ejecución de mínimos cambios en configuraciones ya construidas, la segunda de las opciones analizadas (ros_buildfarm) es bastante más sólida. Ha demostrado ser funcional en un proyecto de mayor magnitud que el abordado por este documento, como es la infraestructura operativa que tiene un proyecto ampliamente extendido como ROS.

Un sistema programado en python que es capaz de crear, modificar y mantener todas las configuraciones de los trabajos en Jenkins necesarios de manera programática (empleando código) pudiera ser una solución perfectamente válida al reto planteado en este punto de mejora del documento.

Considerando esta segunda aproximación como eficaz para la solución del problema también es necesario poner en relieve sus limitaciones y problemas. Al ser una implementación externa totalmente ajena a cualquier soporte oficial de Jenkins, la consistencia del código XML generado con la configuración de las distintas versiones del servidor no está garantizada y el código requerirá cambios si estas diferencias ocurren durante el ciclo de desarrollo futuro de Jenkins.

También encontramos problemas de mantenibilidad en esta solución cuando nuevos plugins son añadidos a Jenkins. Esto requerirá de un esfuerzo de adaptación tanto de la generación en python como de las plantillas XML escritas en python empy.

7.4.1 Solución final adoptada

La última de las soluciones analizadas en la sección de propuestas consiste en emplear una solución si bien no estrictamente nativa del proyecto Jenkins, sí que tiene un alto grado de vinculación al mismo, un grupo de proyectos diversos que apoya su desarrollo, y un plugin que se integra directamente en Jenkins y registra más de 10.000 instalaciones mensuales en Mayo de 2017.

Al comparar las dos soluciones propuestas, la multitud de proyectos que emplean Jenkins DSL comparada con el único empleo de ROS de su propio software, supone un punto a favor a la hora de poder emplear la solución sin necesidad de modificaciones iniciales o adaptaciones.

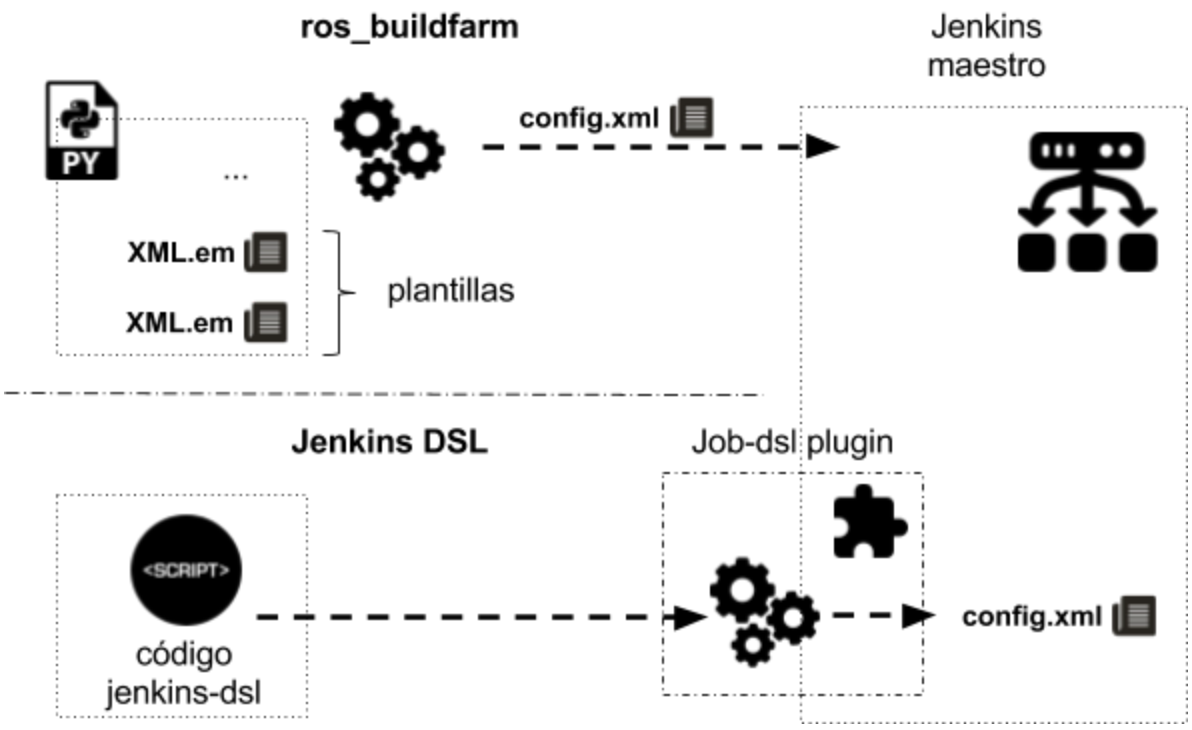


Figura 24: comparativa de generación de configuraciones entre el proyecto ros_buildfarm y jenkins DSL

Quizá el punto más importante a favor de emplear Jenkins DSL sea, como ilustra la figura 24, la posición del mecanismo que implementa la lógica de transformación. Mientras el código del proyecto ROS implementa la lógica totalmente independiente de cualquier librería o herramienta de Jenkins, Jenkins DSL emplea un plugin soportado oficialmente en multitud de versiones del servidor para realizar esta conversión. Este detalle es fundamental para la futura mantenibilidad de la solución a adoptar.

Cabe señalar que cuando el proyecto ROS comenzó a implementar su instalación de Jenkins, el desarrollo de Jenkins DSL no había comenzado. Podría considerarse un esfuerzo paralelo (mismo objetivos) pero realizado al margen de la comunidad Jenkins.

7.4.2 Prototipo inicial

Para probar la capacidad de adaptación de Jenkins DSL para generar las configuraciones necesarias en este proyecto se utilizó un pequeño proyecto no relacionado con el núcleo de las pruebas de Gazebo. Una instalación de un port del proyecto ROS sobre cygwin.

El código del prototipo:

```
job('roscygwin-ci_daily-cygwin64') {
    label('cygwin')

    scm {
        hg('http://bitbucket.org/osrf/release-tools','${RTOOLS_BRANCH}')
    }
    triggers {
       cron('@daily')
    }
    parameters {
       stringParam('RTOOLS_BRANCH','default','release-tool branch to use')
    }
    steps {
       systemGroovyCommand("build.setDescription('RTOOLS_BRANCH: ' +
                           build.buildVariableResolver.resolve('RTOOLS_BRANCH'));")
       shell("jenkins-scripts/cygwin/_ros1_roscygwin_compilation.bash")
    }
    publishers {
       textFinder(/failed/, '', true, false, false)
    }
    wrappers {
        colorizeOutput()
    }
}
```

Explicado cada comando empleado en el prototipo:

- **job:** creación de un trabajo de Jenkins utilizando su nombre como parámetro
- **scm:** gestión del sistema de control de versiones que obtiene el código.
    - **hg:** se emplea mercurial para obtener el repositorio de código que contiene los scripts que generan el entorno de pruebas detallado en la mejora número 1 de este documento. Nótese que el segundo parámetro (indica la rama del repositorio mercurial a obtener) figura la variable *RTOOLS_BRANCH* que está definida como uno de los parámetros del trabajo Jenkins y se define en la sección *parameters.*
    - **triggers:** sección que define qué eventos lanzan la ejecución del trabajo
        - cron: el comando define que el trabajo es ejecutado periódicamente. El valor empleado *@daily* hará que se ejecute una vez al día.
    - **parameters:** sección define parámetros de entrada al invocar el trabajo
        - **stringParam:** se define un parámetro en forma de cadena de texto que podrá ser empleado por el resto del script y configuración. El primer argumento es el nombre de la variable (*RTOOLS_BRANCH*), el segundo es valor por defecto (la rama *default*) y el tercero la descripción que aparecerá en la interfaz web al lanzar el trabajo.

- **steps:** sección que define los pasos que ejecutará el trabajo de Jenkins.
  - **systemGroovyCommand:** script en Groovy que accede directamente a la descripción del trabajo de Jenkins para indicar que rama del repositorio se está empleando.
  - **shell:** ejecuta un comando bash en linux, en este caso invoca el script correspondiente (ver anexo 7).
- **publishers:** sección que archiva los artefactos generados durante el trabajo de Jenkins
  - **textFinder:** parser del resultado en consola para detectar si en el mismo existe la expresión regular /failed/. Modificará el estado final del trabajo a FAIL si se encuentra.
- **wrappers:** sección dedicada a acciones ejecutadas antes o después de la fase de ejecución-
  - **colorizeOutput:** colorea la sintaxis del resultado obtenido por pantalla mediante el empleo de códigos HTML correspondientes a los códigos de color en una shell.

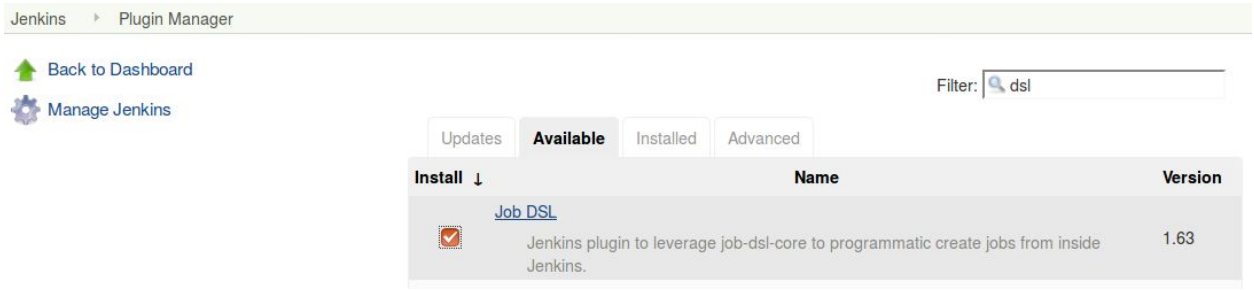


Figura 24: instalación del plugin necesario para procesar Jenkins DSL

El código DSL basado en Groovy requiere de la instalación del plugin de Jenkins correspondiente (ver figura 24) y la creación de un trabajo en el propio servidor, del tipo FreeStyleJob, y configurado en su sección Build para procesar código DSL (ver figura 25).

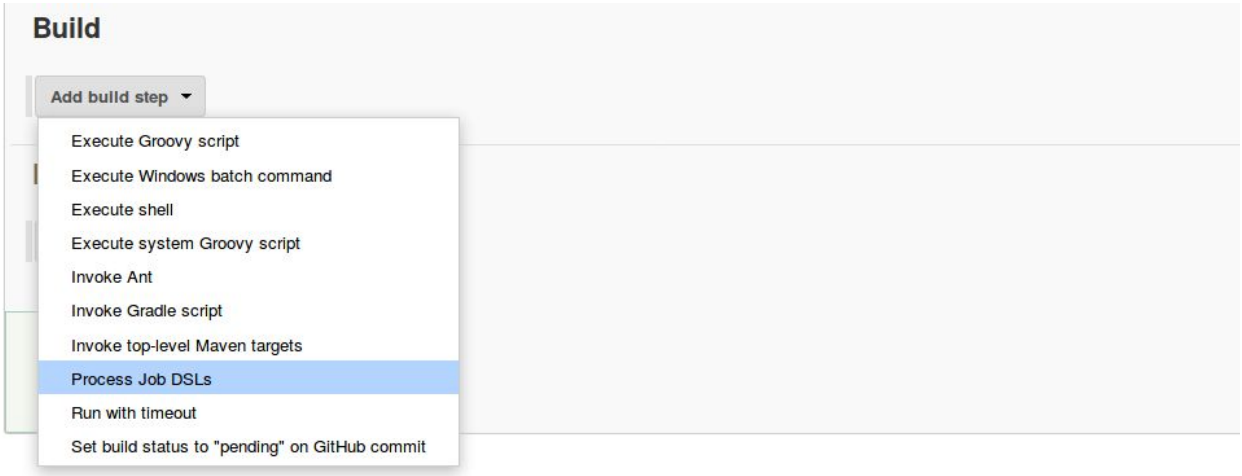


Figura 25: configuración de un trabajo para procesar código Jenkins DSL

En el desplegable que se ofrece se introduce el código DSL directamente (ver figura 26).

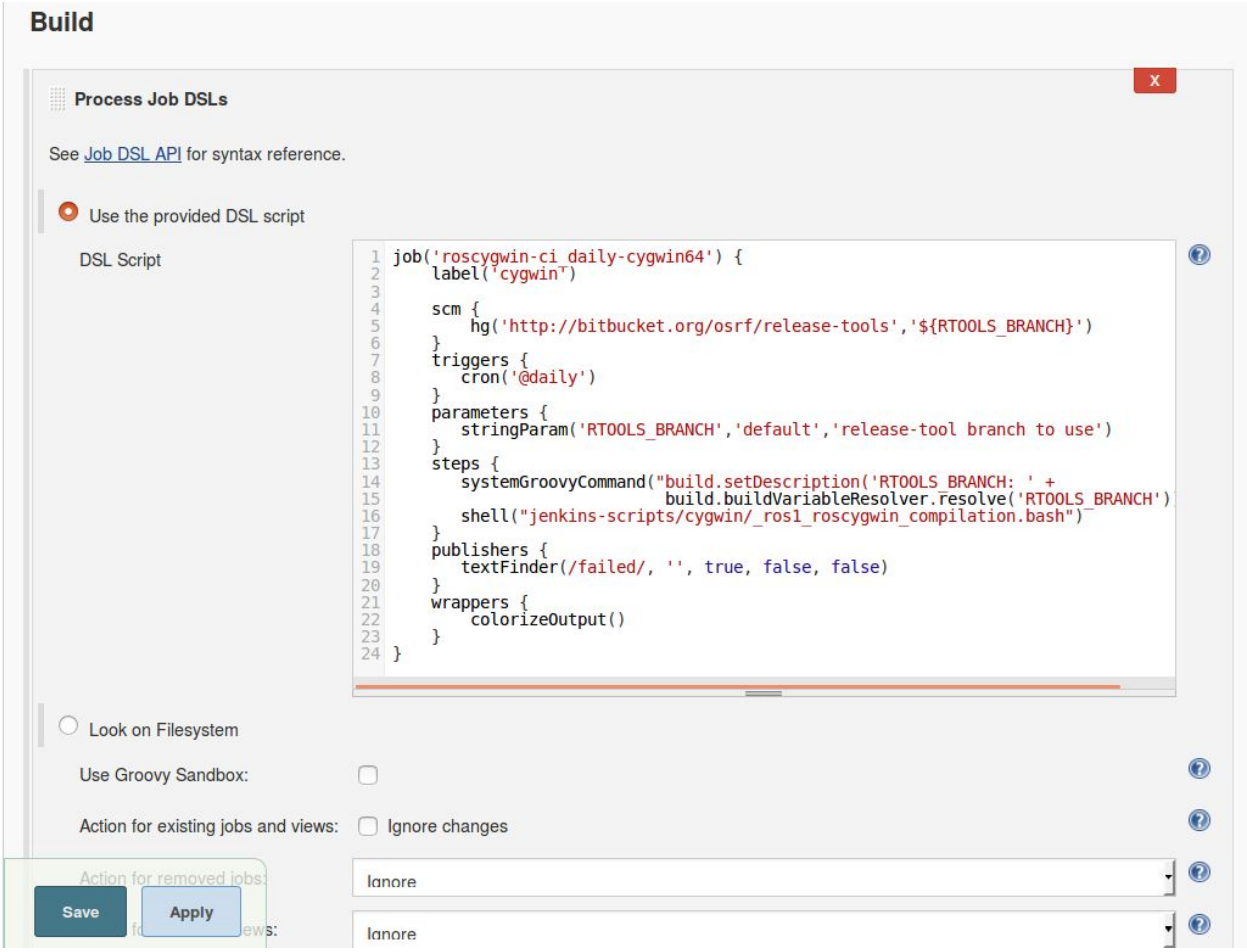


Figura 26: introducción de código en el formulario del plugin Job DSL

La configuración se guarda y el trabajo se ejecuta. La salida que obtenemos por pantalla del proceso de este código Groovy DSL (ver figura 27) es especialmente interesante. El resultado del trabajo es “UNSTABLE” y esto es debido a que el propio trabajo informa de que faltan plugins por instalar en el servidor (*text-finder* y *ansicolor*). Este hecho **refuerza la decisión tomada de elegir Jenkins DSL** en lugar de una implementación externa que genere XML ya que en está última hubiera sido imposible conocer si están todos los plugins necesarios instalados o no con el correspondiente riesgo de terminar con una configuración incompleta.

Figura 27: resultado mostrado por Jenkins tras procesar el código Jenkins DSL del prototipo

La propia salida de texto del trabajo informa sobre el nuevo trabajo generado y su nombre (ver figura 28). Al consultar la lista de trabajos del servidor encontramos el trabajo creado y la indicación del trabajo original que lo creó (*seed job*).

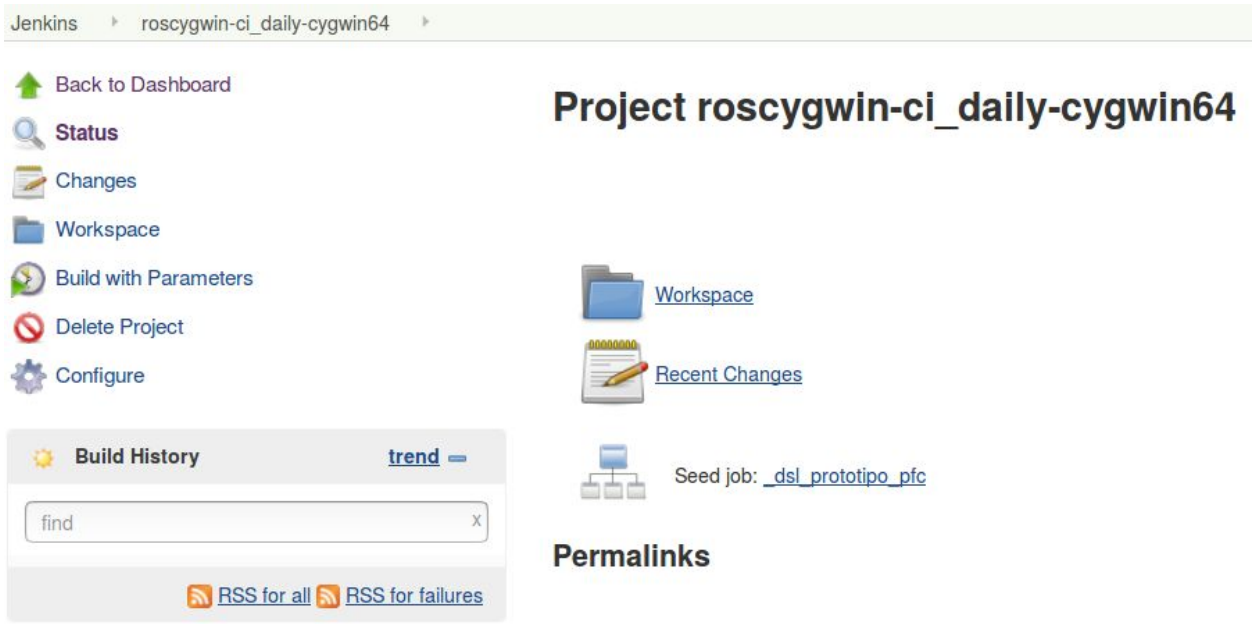


Figura 28: trabajo creado procesar el código Jenkins DSL del prototipo

Una de las herramientas que ha construido la comunidad en torno a Jenkins DSL es una sencilla aplicación web llamada **Jenkins DSL Playground** [23] que permite visualmente comprobar el XML de configuración resultante de procesar un determinado Jenkins DSL (ver anexo 6 para observar la salida de esta herramienta al procesar el código del prototipo).

7.4.3 Reutilización de código: clases Jenkins DSL

Lo simple del prototipo ha permitido confirmar algunas de las premisas descritas en el punto de objetivos pero es una solución incompleta si tenemos en cuenta el global de los requisitos. Si se imagina un servidor con cinco trabajos de configuración similar a la expuesta en el prototipo, la capacidad de **no duplicar este código Jenkins DSL** aparece rápidamente como una necesidad imperiosa para obtener una solución completa.

La ingeniería del software desde sus primeros pasos ha tenido el evitar la duplicidad de código como uno de los objetivos principales a conseguir, con todas las ventajas que ello conlleva. En este caso disponemos de un lenguaje (Groovy) que implementa tanto el paradigma estructurado como la orientación a objetos.

En el diseño de los tipos de trabajos de lo que puede disponer el servidor de pruebas automáticas de la Open Source Robotics Foundation se podría establecer una jerarquía entre los mismos en función de diferentes conceptos. Una base común con los elementos que se quiere estén presentes en todos los trabajos, un segundo nivel que implemente las peculiaridades del sistema operativo al que va dirigido el trabajo, un tercer nivel en que aparezcan los tipos diferentes de trabajos según su función (compilación, generación de paquetes, etc.).

Ejemplificando la teoría con código real, se dispondría de una clase base llamado *OSRFBase* que implementa operaciones comunes a todos los trabajos, como pueden ser el enviar un correo con el resultado del trabajo, incluir un mensaje de aviso en todos los trabajos para que otros usuarios que interactúen con ellos no modifiquen las configuraciones utilizando el interfaz web (serán sobrescritas en la próxima ejecución del trabajo de DSL que genera la configuración). El código que representa esta idea es sencillo:

```
class OSRFBase
{
   static void create(Job job)
   {
     job.with {
       description 'Automatic generated job by DSL jenkins. Please do not edit manually'

        publishers {
          extendedEmail('$DEFAULT_RECIPIENTS, admin-jenkins@osrfoundation.org',
                        '$DEFAULT_SUBJECT',
                        '$DEFAULT_CONTENT')
        }
     }
   }
}
```

Heredando de esta clase Base encontramos la configuración común para entornos Linux. En este caso se necesita indicar que el trabajo solamente puede ser utilizado por nodos que dispongan de docker (comando *label*), se incluye también el código necesario para obtener el repositorio de scripts que contiene el código de creación del entorno de pruebas y ejecución de las mismas junto con la variante de poder usar una rama específica de este repositorio (la explicación detallada se encuentra en la sección de prototipo de este mismo capítulo) y también se especifica la conversión de los códigos de colores shell a HTML para su correcta visualización:

```
class OSRFLinuxBase extends OSRFBase
{
 static void create(Job job)
 {
   OSRFBase.create(job)
   job.with
   {
     label "docker"

     parameters { stringParam('RTOOLS_BRANCH','default','release-tool branch to use') }

     steps
     {
       systemGroovyCommand("build.setDescription('RTOOLS_BRANCH: ' +
                           build.buildVariableResolver.resolve('RTOOLS_BRANCH'));")

       shell("""
          [[ -d ./scripts ]] &&  rm -fr ./scripts
          hg clone http://bitbucket.org/osrf/release-tools scripts -b \${RTOOLS_BRANCH}
          """)
     }

     wrappers {
       colorizeOutput()
     }
   }
  }
}
```

El tercer nivel de la jerarquía sería el destinado a los tipos de trabajos concretos. Inicialmente se dispone de trabajos de compilación (y ejecución de las pruebas automáticas) que llevan asociadas una serie de características de configuración: la prioridad con la que son ejecutados cuando existen varios trabajos en espera (comando *priority*), almacenar el registro de las últimas 15 ejecuciones (comando *logrotator*), activar el reporte por avisos generados por el compilador GCC4 (comando *warnings*) e incluir el mail a enviar a los usuarios el resultado de los tests.

En este nivel existen plugins para los que Jenkins DSL no dispone de implementación propia. En este caso son implementados generando código XML desde el propio lenguaje. Es el caso del reporte de las pruebas automáticas con formato JUNIT o el reporte también del resultado del analizador de código estático cppcheck.

```
class OSRFLinuxCompilation extends OSRFLinuxBase
{
  static void create(Job job)
  {
    OSRFLinuxBase.create(job)

    def mail_content ='''
     $DEFAULT_CONTENT
     Test summary:
     -------------
     * Total of ${TEST_COUNTS, var="total"} tests : ${TEST_COUNTS, var="fail"} failed and
       ${TEST_COUNTS, var="skip"

     Data log:
     ${FAILED_TESTS}
  '''

  job.with
    {
      priority 100

      logRotator {
        numToKeep(15)
      }

      publishers
      {
         // compilers warnings
         warnings(['GNU C Compiler 4 (gcc)'])

         // special content with testing failures
         extendedEmail('$DEFAULT_RECIPIENTS, admin-jenkins@osrfoundation.org',
                       '$DEFAULT_SUBJECT',
                        content)
         // junit plugin is not implemented. Use configure for it
         configure { project ->
            project / publishers << 'hudson.tasks.junit.JUnitResultArchiver' {
                 testResults('build/test_results/*.xml')
                 keepLongStdio false
                 testDataPublishers()
            }
         }
         // cppcheck is not implemented. Use configure for it
         configure { project ->
           project / publishers / 'org.jenkinsci.plugins.cppcheck.CppcheckPublisher' /
```

```
            cppcheckConfig {
              pattern('build/cppcheck_results/*.xml')
              ignoreBlankFiles true
              allowNoReport false
            }
          }
        }
      }
    }
}
```

Con este diseño de clases la implementación de la configuración para el trabajo estandard de compilación y generación de pruebas del proyecto Gazebo en su ciclo de integración continua quedaría simplificado a:

```
// Create the default ci jobs
def ci_default_job = job("gazebo-default-devel-precise-amd64”)

// Use the linux compilation as base
OSRFLinuxCompilation.create(ci_default_job)

ci_default_job.with
{
  scm {
   hg('http://bitbucket.org/osrf/gazebo')
  }

  triggers {
    scm('*/5 * * * *')
  }

   steps {
    shell("""
          export DISTRO=precise
          export ARCH=amd64
          /bin/bash -x ./scripts/jenkins-scripts/gazebo-default-gui-test.bash")
   }
}
```

Nota: el evento que dispara la compilación (configura en *triggers*) está configurado para comprobar el repositorio de Gazebo en busca de cambios en la rama principal cada 5 minutos y disparar el trabajo si encuentra modificaciones desde la última ejecución. Emplea la misma sintáxis que el software *cron*.

Para comprobar otro de los aspectos que figuran en los objetivos como es de la **escalabilidad,** se implementa la opción de generar un trabajo de compilación sobre la rama principal del repositorio (*default*) en dos distribuciones de Ubuntu Linux (Precise y Trusty) y dos arquitecturas (amd64 e i386).

El ejemplo previo solamente implementa Ubuntu Precise sobre amd64, aquí se añade i386, pero sería trivial extender el array para que soporte alguna variante de ARM.

```
def supported_distros = [ ‘precise’, 'trusty' ]
def supported_arches = [ 'amd64', ‘i386’ ]

supported_distros.each { distro ->
  supported_arches.each { arch ->

    // Create the default ci jobs
    def ci_default_job = job("gazebo-default-devel-${distro}-${arch}")

    // Use the linux compilation as base
    OSRFLinuxCompilation.create(ci_default_job)

    ci_default_job.with
    {
       scm {
         hg('http://bitbucket.org/osrf/gazebo’)
       }

       triggers {
         scm('*/5 * * * *')
       }

       steps {
         shell("""
           export DISTRO=${distro}
           export ARCH=${arch}
           /bin/bash -x ./scripts/jenkins-scripts/gazebo-default-gui-test.bash")
      }
    }
  }
}
```

La modificación para ampliar el número de arquitecturas y distribuciones soportadas ha consistido en definir dos listas con los nombres de las mismas, utilizar dos bucles each para iterar sobre las mismas y parametrizar aquellos valores necesarios.

Añadir una nueva arquitectura o una nueva distribución de Ubuntu sería tan sencillo como añadirla a la lista correspondiente. Eliminar una distribución cuando cesa su soporte oficial es tan sencillo como eliminarla de la lista y ejecutar de nuevo el código DSL.

Podríamos definir un “diagrama de clases” conceptual que ayude a entender la jerarquiía expuesta en esta sección de una manera clara y visual (ver figura 29)

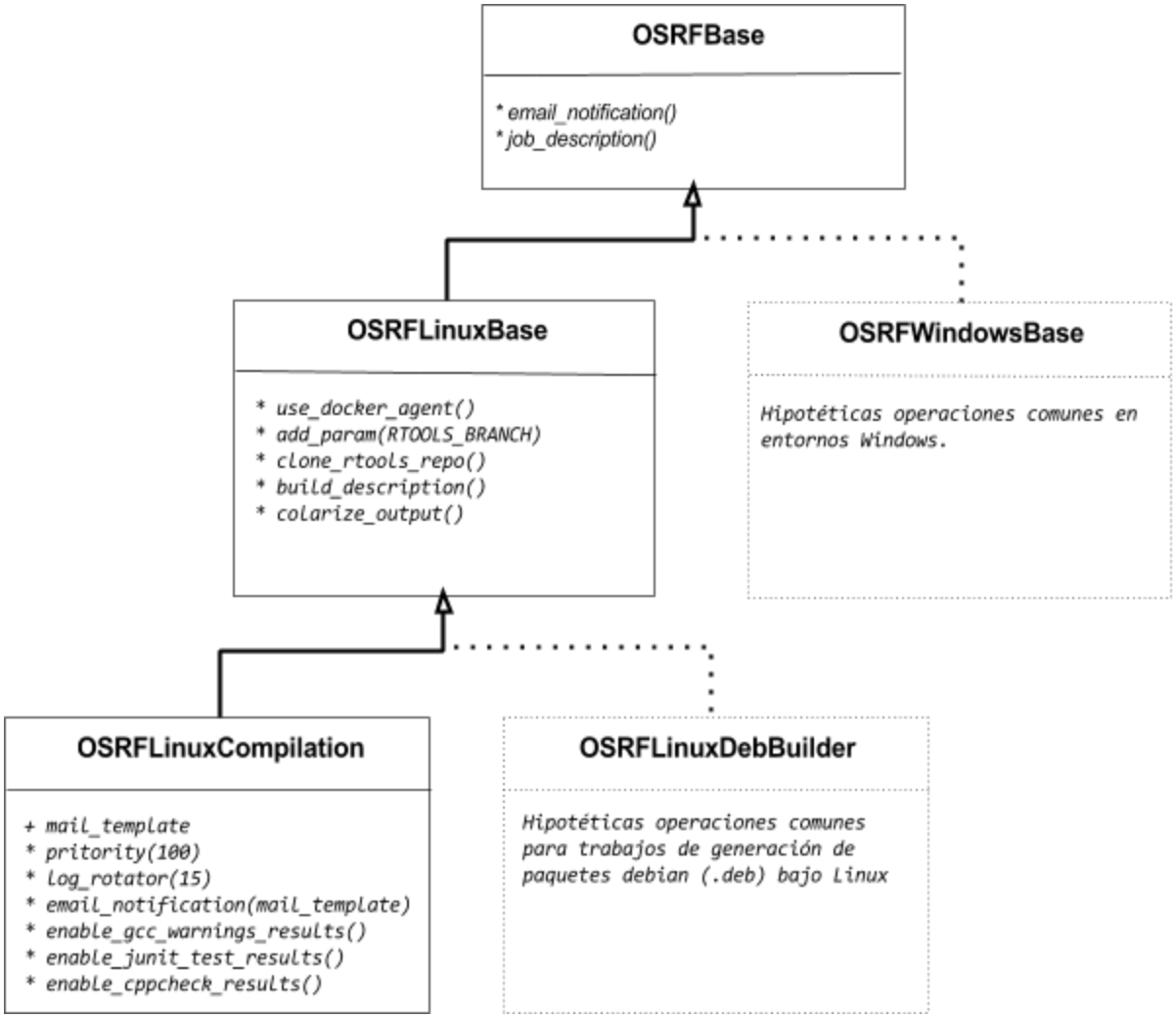


Figura 29: Diagrama de clases existentes e hipotéticas (línea discontinua) de Jenkins DSL para la jerarquía existente entre los trabajos

Capítulo 8

8. Presupuesto

Este capítulo aborda el cálculo económico y socioeconómico del proyecto. Está dividido cuatro secciones: la primera, es el cálculo del presupuesto para personal, basado en la estimación temporal de las diferentes tareas. La segunda aborda el coste de los sistemas informáticos involucrados en la infraestructura descrita por el proyecto. La tercera estima el coste total del proyecto y por última la cuarta sección reflexiona sobre los beneficios sociales y el impacto ambiental de los cambios propuestos.

8.1 Coste de personal para desarrollo e implementación

Para el cálculo del coste de desarrollo es necesario el desglose de las tareas que componen el proyecto así como la estimación del tiempo empleado en las mismas. Presuponiendo que las personas que llevan a cabo el desarrollo tienen conocimiento medios sobres sistemas Linux y procesos de integración continua:

Trabajo de documentación previo:

Tarea	Días/Hombre
Adquirir conocimientos avanzados sobre Jenkins	20
Adquirir conocimientos medios sobre chroot	10
Adquirir conocimientos medios sobre Gazebo y sus sitema de pruebas automáticas	5

Subproyecto I: mejora y control del código de generación entornos de pruebas

Tarea	Días/Hombre
Adquirir conocimientos sobre soluciones alternativas: plugins de Jenkins	2
Implementación de la solución: creación del repositorio e integración en los scripts	2

Subproyecto 2: soporte para la ejecución de pruebas con GPU

Tarea	Días/Hombre
Adquirir conocimientos sobre aceleración gráfica dentro de entornos chroot	2
Adquirir conocimientos sobre soluciones alternativas: servidores gráficos virtuales	4
Implementación de detección de soporte de aceleración gráfica e integración en cmake	5
Implementación de soporte para la réplica del stack gráfico del sistema anfitrión en el chroot	5

Subproyecto 3: mejora del entorno virtual de pruebas

Tarea	Días/Hombre
Adquirir conocimientos medios sobre soluciones alternativas: virtualización, KVM, VirtualBox, LXC, Docker	15
Adquirir conocimientos avanzados sobre Docker	10
Migración de scripts desde chroot a Docker	10

Subproyecto 4: mejora del entorno virtual de pruebas

Tarea	Días/Hombre
Adquirir conocimientos medios sobre soluciones alternativas: ros_buildfarm	5
Adquirir conocimientos avanzados sobre Jenkins DSL	7
Implementación clases y scripts DSL	10

Total de días hombre: 112 días/hombre = 896 horas/hombre

Presupuesto suponiendo un/a ingeniero/a senior 4.289,54 € hombre/mes. Implicando 1 hombre mes un máximo de 131,25 horas, tendríamos una duración casi 7 meses.
Coste aproximado de personal para la ejecución de este proyecto asciende a 37.833,74 euros.

8.2 Sistemas informáticos

Para la implementación de un entorno de integración continua como el descrito en este proyecto existe una necesidad material de equipos informáticos que puedan alojar el tanto el servidor central donde residirá el “master” de Jenkins como los diferentes nodos que ejecutan las pruebas automáticas.

No es necesario que sean sistemas extraordinariamente potentes pero hay factores que influyen muy directamente en que los tiempos sean menores y existen algunos requerimientos mínimos para llevar a cabo la compilación de un sistema C++ complejo como es Gazebo. En el apartado de requerimientos mínimos podríamos destacar la cantidad de memoria RAM que utiliza el compilador C++, especialmente cuando se lanzan varios hilos de compilación en paralelo. Un sistema mínimo necesitaría de no menos de 8Gb para mantener 2 hilos de compilación y unos 16Gb para poder lanzar al menos 4. También disponer de un procesador multi-thread ayudará a reducir los tiempos y un factor clave, dada la cantidad de dependencias y cabeceras que entran en juego en la compilación, puede ser disponer de una tasa alta de entrada/salida al disco. Los nodos que disponen de discos SSD resultan mucho más eficientes.

Sería deseable disponer de dos agentes para Jenkins provisto con las características descritas en el párrafo anterior:

Función	Detalles	Precio
Nodo Nvidia	Dell Alienware Aurora: *Procesador Intel Core™ i7-6700, NVIDIA GeForce GTX 950 Memoria DDR4 no ECC de 16 GB (2 x 8 GB) a 2133 MHz, SSD PCIe de 256 GB + unidad SATA a 6 Gb/s de 1 TB a 7200 rpm [24]*	1.269 € por unidad

Para el master de Jenkins (no realiza compilaciones, únicamente ejecuta el servidor) se puede optar por no mantener una máquina local y disponer de una instancia de Amazon. El único requerimiento es una cantidad de RAM suficiente para gestionar todos los plugins y trabajos del propio servidor de Jenkins.

Función	Detalles	Precio
Sistema para el servidor Jenkins (master)	Master Amazon EC2 t2.large (8Gb)	0.108/hora, 946€ anuales

Para el cálculo del gasto anual total hay que computar la amortización de los dos sistemas NVIDIA. No existiría amortización sobre el sistema de Amazon. El resumen total de gasto sería:

Concepto	Coste
2 x Sistema Dell Alienware	1.269€ + 1.269€ a amortizar en 60 meses. 507€ anuales en total.
1 x Amazon EC2	946€
Total anual en gasto informático	1453€

8.3 Coste Total

Para el coste total del proyecto se procede a sumar el coste del personal, la amortización de los sistemas empleados por Jenkins, una pequeña estimación de costes de funcionamiento (cables ethernet, repuestos hardware, ordenador de desarrollo, etc.) de unos 1200€. Además se considera una tasa de costes indirectos de un 20%.

Concepto	Valor
Personal	37.833,74€
Amortización	1.453€
Costes funcionamiento	1.200€
Costes indirectos 20%	8.097,34€
Total proyecto	**48.584,08€**

8.4 Impacto socioeconómico

El marco de desarrollo del proyecto, una serie de mejoras sobre un proyecto de código abierto que no figura bajo el abanico de una financiación con objetivos concretos, hacen que no sea trivial estimar el impacto socioeconomómico del mismo. Aunque los cálculos concretos resulten muy complejos, sí que se pueden detallar una serie de aspectos relevantes.

Cabría destacar que Gazebo es un proyecto de código abierto y gratuito empleando por aficionados a la robótica, empresas (ej: Pal Robotics Barcelona), universidades (ej: Shanghai) y concursos de robótica internacionales (ej: Robocup) de todas partes del mundo. Las mejoras descritas en este documento impactan en la mejora de la calidad del propio proyecto, así como en las versiones que estarán a disposición de todos los usuarios anteriormente mencionados. Existe por tanto un **impacto económico distribuido a nivel mundial**, especialmente para las iniciativas profesionales y universidades que se benefician a coste cero.

Todos los usuarios de Gazebo pueden beneficiarse sin ningún tipo de barrera económica del resultado de las mejoras descritas en este documento (versiones de Gazebo con mayor calidad). El efecto de proporcionar potentes herramientas software a coste cero a nivel global que tienen los proyectos de código abierto, repercute directamente como **beneficio social** a todos los ámbitos de la humanidad que los emplean, en el caso de Gazebo desde proyectos de la NASA hasta usuarios que trabajan en la mejora de la logística [25] distribuida.

Aunque el proyecto no tiene un impacto claro o medible desde un punto de vista **medioambiental**, alguna de las mejoras, como la número 2 (activación del soporte para pruebas que emplean la GPU) empeoran la situación inicial, ya que se emplearía mayor energía eléctrica y un mayor desgaste del hardware en cada iteración del bucle de integración continua. Otras impactarían positivamente con los mismo motivos, como la mejora número 3 en su vertiente de reducir los tiempos empleados en cada ejecución.

Focalizando en el entorno de la Open Source Robotics Foundation (OSRF), existe un impacto sobre el prestigio del propio software, al aumentar su calidad, y por tanto, también un impacto en el prestigio e imagen de la propia fundación, que tendrá un poco más sencillo la consecución de nuevos proyectos basados en el simulador. Existe también una reducción de costes para la OSRF ya que las mejoras 1 (control de código en repositorio) y 4 (automatización de la configuración de Jenkins) ahorran muchas horas de trabajo de creación y sobre todo mantenimiento. También la mejora número 3 (reducción del tiempo y mejora del aislamiento del entorno de pruebas) produce claramente un impacto económico positivo al tener que emplear menos tiempos en ejecución, lo que se podría traducir por un menor coste de servidores en la nube o un menor gasto de hardware y electricidad para servidores alojados de manera propia.

Capítulo 9

9. Conclusiones generales

Este último capítulo describe las conclusiones generales sobre varios puntos que afectan a uno o varios de los subproyectos detallados en el mismo.

Las diferentes mejoras sobre la situación inicial en que se encontraba el proceso de integración continua ponen de relieve detalles interesantes que se podrían extrapolar probablemente a un gran conjunto de aplicaciones dentro de la misma categoría que el simulador Gazebo.

Algunas de las decisiones analizadas en este documento, especialmente en los primeros capítulos o mejoras, se remontan varios años atrás desde la fecha de escritura de este capítulo de conclusiones. Esto permite realizar una mirada retrospectiva crítica y realizar algunas afirmaciones basadas en hechos que ocurrieron posteriormente a su implementación.

9.1 Elección de jenkins como servidor

La decisión de mantener la instalación inicial de Jenkins y optar por él como servidor central de integración continua fue probablemente un acierto. Con el paso de los años Jenkins ha ido ganando usuarios, desarrolladores y la empresa que soporta el equipo principal de desarrollo (CloudBees) se ha consolidado. El crecimiento del número de trabajos (de integración continua) a nivel mundial, el número total de plugins y el número de nodos han sufrido crecimientos exponenciales.

Desde la aparición de los servicios integrados de testing en la nube (travis, circle, drone, etc.) y especialmente su integración con los principales servicios de hosting de código (github, bitbucket, etc.) muchos proyectos open source están actualmente utilizándolos como servidores de integración continua. Conviene destacar aquí que estos servicios no estaban disponibles cuando este proyecto comenzó, aparecieron durante la ejecución de los diferentes subproyectos.

Existen algunos factores intrínsecos al proyecto que harían que el reemplazo de jenkins por este tipo de servicios fuera complejo o muy costoso :

1. Tiempo de compilación y ejecución de los tests: servicios como travis limitan el tiempo máximo que un usuario puede ocupar una de sus instancias virtuales de test.
2. Disponibilidad de un servidor X y una GPU: en el caso de los tests gráficos de los que dispone Gazebo, habría que estudiar la posibilidad de utilizar un servidor X virtual (xnest, Xvfb, etc.) En muchos de los servicios online esta opción no es viable. La opción de necesitar una GPU complica mucho más que pudieran ser una alternativa válida ya que no existen opciones que lo permitan.
3. Multi-platforma: el gran número de versiones de Ubuntu/Debian que soporta Gazebo, el soporte MacOSX a través de Brew o la posibilidad de añadir un soporte para Windows complicarían enormemente el empleo de estas opciones.

Existen también puntos de contacto y retroalimentaciones desde los proyectos de servicios en la nube hacia Jenkins y viceversa. Por ejemplo, el propio proyecto Jenkins ha implementado soporte para la existencia de un Jenkinsfile, que sigue la filosofía de lenguaje declarativo (declarative programming), muy similar al fichero de configuración de travis, .travis.yml.

9.2 El boom de los “contenedores”: docker

Como punto fuerte destaca también la toma de decisión, dentro del marco de este proyecto, que fue la selección del entorno de testing que debería reemplazar al chroot inicial del que se disponía. Si la decisión de utilizar Jenkins vino respaldada por un crecimiento exponencial en los meses siguientes, el caso de docker alcanza cotas aún mayores, con el punto de inflexión a principios de 2015:

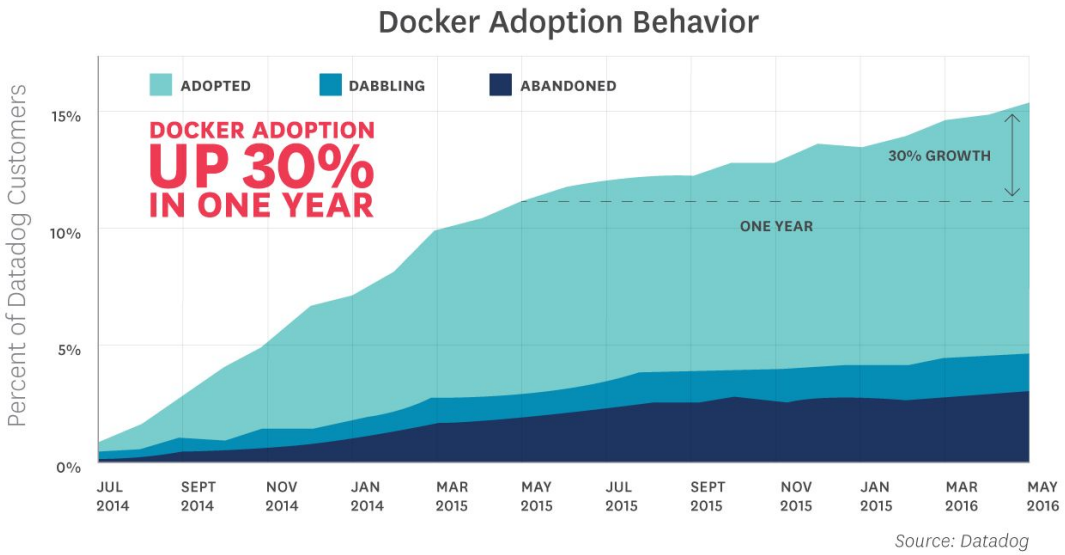


Figura 30: crecimiento de Jenkins durante 2016

El desarrollo de docker desde el año 2015 hasta la fecha resulta espectacular y el número de proyectos actualmente utilizándolo también.

El desarrollo de un software de orquestación para clusters (Docker Swarm) o la implementación utilizando un lenguaje declarativo (Docker Compose) son algunas de las herramientas potentes desarrolladas por el proyecto docker que entrarían en el apartado de trabajos futuros para complementar el despliegue actual mostrado en este proyecto.

9.3 Configuración desde código: jenkins-dsl

La elección de empleo de jenkins-dsl tiene una doble lectura en clave interna para la Open Source Robotics Foundation. Una parte es positiva: generar configuración desde código soportado de forma nativa por Jenkins y su equipo de desarrollo surge en este año 2017 como la mejor forma de conseguir el objetivo propuesto y el proyecto no ha parado de tener contribuciones y extensiones a los nuevos plugins que van surgiendo. Otra lectura en clave interna de la OSRF es menos positiva: la divergencia en la manera de implementación entre proyectos cercanos, como son ROS (Robot Operative System) y Gazebo, impide la colaboración y contribución dentro de un misma plantilla profesional y desarrolladores muy cercanos a la hora de implementar un mismo objetivo.

9.4 Estado de la granja de pruebas en 2017

Todas las acciones descritas en el presente documento fueron puestas en prácticas y han sido desarrolladas dentro de la granja de automatización de la que dispone la Open Source Robotics Foundation y que es de acceso público: http://build.osrfoundation.org/

En Julio de 2017 el servidor Jenkins disponía de 8 agentes, en plataformas tan dispares como Ubuntu, Debian, MacOSX, Windows y llegó a disponer incluso de algún trabajo en Cygwin. Hasta ocho versiones diferentes de Ubuntu desde su comienzo han sido soportadas y en la actualidad tres versiones de MacOSX están en uso. Existen trabajos en arquitecturas x86 (amd64 e i386) y ARM (arm64 y armhf). **El número total de trabajos actuales es de 540**, al inicio de este proyecto existían menos de 10.

La generación de la configuración sigue empleando **jenkins-dsl**, existe un trabajo dsl por cada tipo de proyecto que genera todos los trabajos relativos al mismo. Se pueden observar todos los trabajos desde la interfaz del trabajo global que actualiza todos cada día:
https://build.osrfoundation.org/job/_dsl_generate_all_dsls/

Todos los entornos virtuales son generados utilizando **docker** y la OSRF dispone de imágenes precompiladas para Gazebo disponibles desde el docker Hub: https://hub.docker.com/_/gazebo/. No solamente son empleadas en tareas de pruebas, existe un buen número de usuarios que ante la gran cantidad de dependencias que exige la instalación estándar optan por ejecutar Gazebo directamente desde docker.

Capítulo 10

10. Futuras lıneas/trabajos

Abordar un proceso de pruebas automáticas de un simulador de robótica implica un buen número de líneas de trabajo en diferentes ámbitos. Cada una de estas líneas abre de cara al futuro múltiples posibilidades tanto por el abanico de opciones existentes, como por los desarrollos y las opciones que se implementa en el presente y que hacen cambiar el espectro de opciones donde elegir. Este documento detalla algunos aspectos importantes pero en el futuro hay opciones interesantes que se podrían explorar:

En el apartado de empleo de la **GPU en entornos virtuales de pruebas automáticas**, existe actualmente un proyecto desarrollado por NVIDIA que proporciona a docker la capacidad de empleo del sistema de aceleración gráfica del entorno anfitrión de manera sencilla. El proyecto recibe el nombre de nvidia-docker y es un wrapper sobre el propio comando docker (nvidia-docker es también el nombre del comando) que permite emplear fácilmente con la misma interfaz que el propio docker. Tiene la inmensa ventaja de ser una solución portable que hace las imágenes agnósticas respecto del driver nvidia empleado. El impacto directo de esta funcionalidad es la capacidad de ejecutar cualquier versión Linux en el ciclo de integración continua sobre cualquier nodo independientemente de que el sistema anfitrión y el entorno virtual sean el mismo.

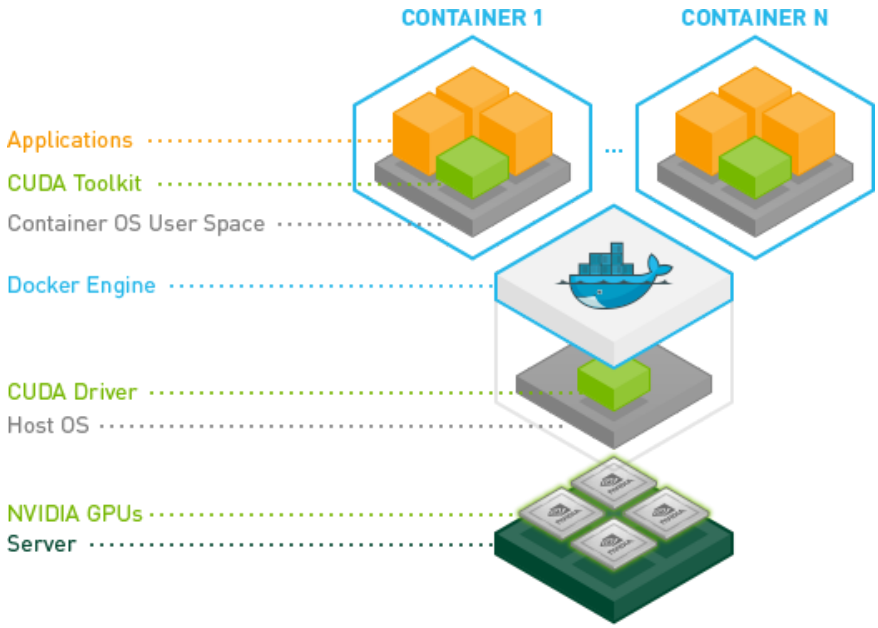


Figura 31: capas lógicas empleadas por el proyecto nvidia-docker

En el ámbito de los entornos virtuales, con **docker** como opción principal durante todo el año 2017 y en pleno crecimiento, no se antoja una posibilidad cercana de cambiar hacia una tecnología o implementación nueva en el corto plazo. Donde sí se tendría una posibilidad de mejora amplia empleando docker sería en compartir la caché de la que dispone cada nodo, de esta manera podrían ahorrar esfuerzos en las primeras ejecuciones que generan caché.

En el área de generación de configuraciones desde código programable para Jenkins, también el proyecto **Jenkins DSL** se encuentra muy activo en 2017. Paralelamente el proyecto Jenkins ha implementado lo que se ha conoce como “Jenkins Pipelines” que implementa el paradigma de programación declarativa y podría emplearse junto con Jenkins DSL para reducir el número de scripts bash necesarios y en la medida de lo posible los esfuerzos de mantenimiento del código que genera la configuración.

Otro de los aspectos que podrían mejorar el tiempo de creación de un entorno virtual sería la instalación de un proxy local que realizará las tareas de almacenamiento de los paquetes debian descargados. Numerosas imágenes utilizan exactamente los mismo paquetes. Existen herramientas linux sencilla, como pueda ser *squid-deb-proxy*, que podrían servir para reducir los tiempos en la descarga de paquetes y mejorar el rendimiento global de las ejecuciones.

Fuera del alcance de este proyecto ha quedado la ampliación realizada a otras plataformas, como Windows o MacOSX, sobre las cuales también resultaría interesante investigar qué herramientas proporcionan las características estudiadas por este proyecto (ej: *apple sandbox* para entornos virtuales en MacOSX, *vckpkg* o *conan.io* como soluciones de instalación de paquetes en Windows)

Capítulo 11

11. Marco regulador

El proyecto desarrollado en este documento trata sobre mejoras realizadas en su mayor parte empleando código abierto con diferentes licencias (ver tabla 5). Ninguno de los aspectos iniciales del proyecto o desarrollados durante el mismo están sujetos a cláusulas de confidencialidad ni a condiciones comerciales particulares de alguna empresa u organismo.

Software	Licencia	Software	Licencia
Gazebo	Apache2	**Jenkins**	MIT
Docker	Apache2	**Jekinks DSL**	Apache2
LXC	GNU GPL2	**Bash**	GPL3

Tabla 5: ejemplos software empleados en el proyectos y licencias de uso

El único software que interviene como parte fundamental del proceso y cuenta con una licencia propietaria son los controladores oficiales de NVIDIA para sus tarjetas gráficas en Linux. Ninguna de las acciones detalladas en este documento va más allá de emplear los controladores para el hecho que fueron diseñados ni produce ningún trabajo derivado empleando los mismos.

No se maneja información confidencial que pudiera requerir del cumplimiento de la legislación vigente en materia de privacidad, tanto en Estados Unidos donde existen una multitud de leyes federales como puede ser el cumplimiento de Federal Trade Commission Act (FTC Act) o en España, donde se aplicaría la Ley Orgánica de Protección de Datos de Carácter Personal (LOPD) entre otras.

Por la naturaleza puramente software (pruebas sobre un simulador) y de objeto genérico del proyecto (mejoras sin aplicación estrechamente relacionada con algún ámbito laboral), tampoco resulta necesario el análisis de otras normativas como puedan ser las leyes relativas a riesgos laborales o seguridad de los trabajadores. Si se hubiera salido de la simulación a entornos reales en el marco legislativo español, una buena fuente de información es la Asociación Española de Normalización y Certificación (AENOR) y en concreto los estándares ISO que afectan al ámbito de la robótica, como pudiera ser el UNE-EN ISO 13482:2014 (Robots y dispositivos robóticos. Requisitos de seguridad para robots no industriales. Robots de asistencia personal no médicos).

Capítulo 12

12. Bibliografía

1. Koenig, Nathan, and Andrew Howard. "Design and use paradigms for gazebo, an open-source multi-robot simulator." *Intelligent Robots and Systems, 2004.(IROS 2004). Proceedings. 2004 IEEE/RSJ International Conference on*. Vol. 3. IEEE, 2004.
2. C. E. Aguero, N. Koenig, I. Chen, H. Boyer, S. Peters, J. Hsu, B. Gerkey, S. Paepcke, **J. L. Rivero**, J. Manzo, E. Krotkov, G. Pratt, "Inside the Virtual Robotics Challenge: Simulating Real-Time Robotic Disaster Response", *IEEE Trans. Autom. Sci. Eng.*, vol. 12, no. 2, pp. 494-506, Apr. 2015.
3. Merkel, Dirk. "Docker: lightweight linux containers for consistent development and deployment." *Linux Journal* 2014.239 (2014): 2.
4. Alliance, S. C. O. P. E. "Virtualization: State of the Art." (2008).
5. Popek, Gerald J.; Goldberg, Robert P. (1974). "Formal requirements for virtualizable third generation architectures". Communications of the ACM. 17 (7): 412–421. doi:10.1145/361011.361073. Retrieved 2015-03-01. (virtualización)
6. Beck, Kent, et al. "Manifesto for agile software development." (2001): 2006.
7. Docker. Julio 2017: https://docs.docker.com/
8. Jenkins DSL. Julio 2017: https://jenkinsci.github.io/job-dsl-plugin/
9. ROS, Graphic aceleration. Julio 2017: http://wiki.ros.org/ROS/Tutorials/InstallingIndigoInChroot
10. Gazebo documentation. Julio 2017: http://gazebosim.org/tutorials

12.1 Autores de los iconos empleados en las ilustraciones

Icons made by http://www.flaticon.com/authors/those-icons from www.flaticon.com

- http://www.flaticon.com/packs/servers-database

Icons made by http://www.flaticon.com/authors/ocha from www.flaticon.com

- http://www.flaticon.com/packs/humanitarian-icons-2

Capítulo 13

13. Referencias

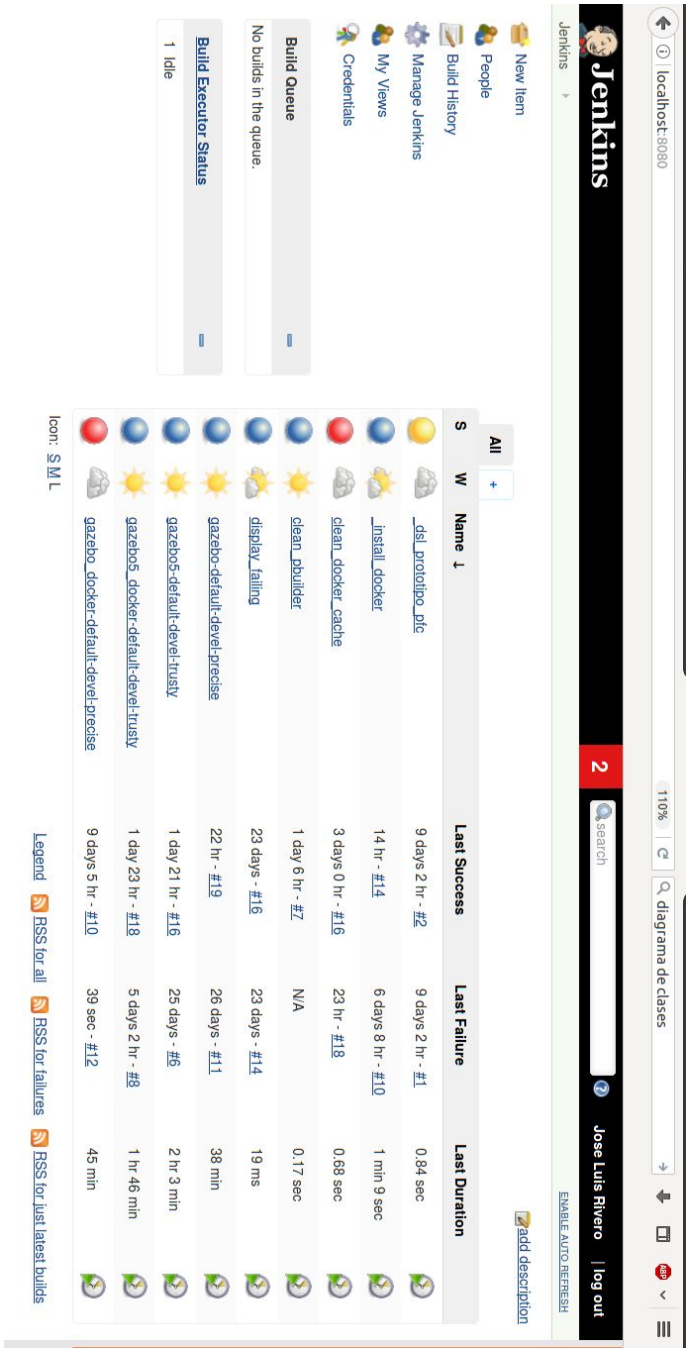
[1] Open Source Robotics Foundation Mission. 2017. Disponible [Internet]: <https://www.osrfoundation.org/about/> [Febrero 2017]
[2] Gazebo Simulator. 2017. Disponible [Internet]: <http://gazebosim.org/> [2013-2107]
[3] Continuous Integration. 2017. Disponible [Internet]: <https://martinfowler.com/articles/continuousIntegration.html> [2013-2017]
[4] Koenig, N. (2012). *Robot life-long task learning from human demonstrations: A bayesian approach. Cres-12-002*. Los Angeles, CA: University of Southern California.
[5] Gazebo: install dependencies from source on Ubuntu. Disponible [Internet]: <http://gazebosim.org/tutorials?tut=install_dependencies_from_source&cat=install>
[6] Gazebo Components.Disponible [Internet]: <http://gazebosim.org/tutorials?tut=components> [2017]
[7] Simulation Description Format. Disponible [Internet]: <http://sfdformat.org> [2017]
[8] Jenkins: managing plugins. Disponible [Internet]: <https://jenkins.io/doc/book/managing/plugins/> [2017]
[9] Languages - Travis CI. Disponible [Internet]: <https://docs.travis-ci.com/user/languages> [2017]
[10] Automated Test Environments for DevOps. Disponible [Internet]: <https://www.capgemini.com/blog/capping-it-off/2017/02/automated-test-environments-for-devops> [2013-2017]
[11] Building and configuring BIND 9 in a chroot jail. Disponible [Internet]: <http://www.unixwiz.net/techtips/bind9-chroot.html> [2013-2017]
[12] The FreeBSD Jail: when chroot is not enough. Disponible [Internet]: <https://www.giac.org/paper/gsec/3807/fredbsd-jail-chroot-enough/102662> [2013-2017]
[13] Linux Container Project. Disponible [Internet]: <https://linuxcontainers.org/> [2017]
[14] Does source code duplication mater?. Disponible [Internet]: <https://solidsourceit.wordpress.com/2012/08/03/does-source-code-duplication-matter/> [2017]

[15] Accessing graphical applications inside the chroot. Disponible [Internet]: <https://help.ubuntu.com/community/BasicChroot#Accessing_graphical_applications_inside_the_chroot> [2017]
[16] A Brief History of Virtualization. Disponible [Internet]: <https://wiki.xenproject.org/wiki/Book/HelloXenProject/1-Chapter#A_Brief_History> [2017]
[18] Ubuntu 15.10: KVM vs. Xen vs. VirtualBox Virtualization Performance. Disponible [Internet]; <http://www.phoronix.com/scan.php?page=article&item=ubuntu-1510-virt&num=5> [2017]
[19] What does Docker technology add to just plain LXC?. Disponible [Internet]: <https://docs.docker.com/engine/faq/#what-does-docker-technology-add-to-just-plain-lxc> [2017]
[20] InfrastructureAsCode, Martin Fowler. Disponible [Internet]: <https://martinfowler.com/bliki/InfrastructureAsCode.html> [2013-2017]
[21] Python empy project. Disponible [Internet]: <http://www.alcyone.com/software/empy/> [2013- 2017]
[22] How We Build Code at Netflix. Disponible [Internet]: <https://medium.com/netflix-techblog/how-we-build-code-at-netflix-c5d9bd727f15> [2017]
[23] Jenkins Job DSL Playground. Disponible [Internet]: <http://job-dsl.herokuapp.com/> [2017]
[24] AlienWare Dell official store. Disponible [Internet]: <http://www.alienware.es/desktops.aspx> [2017]
[25] Simulation of the RoboCup Logistic League with Fawkes and Gazebo for Multi-Robot Coordination Evaluation. Disponible [Internet]: <https://www.fawkesrobotics.org/media/publications/zwilling-thesis-2013.pdf>

Capítulo 14

14. Anexos

14.1 Anexo 1: Interfaz de Jenkins para el proyecto



14.2 Anexo 2: código inicial de ejecución de pruebas automáticas - chroot inicial (pbuilder)

```
#!/bin/bash -x

TIMEDIR=${WORKSPACE}/timing
mkdir -p $TIMEDIR

date +%s > ${TIMEDIR}/absolute_init

######################################################
# Boilerplate.
# DO NOT MODIFY

#stop on error
set -e

distro=precise
arch=amd64
base=/var/cache/pbuilder-$distro-$arch

aptconffile=$WORKSPACE/apt.conf

#increment this value if you have changed something that will invalidate base tarballs. #TODO
this will need cleanup eventually.
basetgz_version=2

rootdir=$base/apt-conf-$basetgz_version

basetgz=$base/base-$basetgz_version.tgz
output_dir=$WORKSPACE/output
work_dir=$WORKSPACE/work

sudo apt-get update
sudo apt-get install -y pbuilder python-empy python-argparse debhelper # todo move to server
setup, or confirm it's there

if [ -e $WORKSPACE/catkin-debs ]
then
  rm -rf $WORKSPACE/catkin-debs
fi

sudo apt-get install --reinstall python-pkg-resources ubuntu-archive-keyring pbuilder
git clone https://github.com/ahendrix/catkin-debs $WORKSPACE/catkin-debs -b master --depth 1


cd $WORKSPACE/catkin-debs
. setup.sh

#setup the cross platform apt environment
# using sudo since this is shared with pbuilder and if pbuilder is interupted it will leave a
sudo only lock file.  Otherwise sudo is not necessary.
# And you can't chown it even with sudo and recursive
sudo PYTHONPATH=$PYTHONPATH $WORKSPACE/catkin-debs/scripts/setup_apt_root.py $distro $arch
$rootdir --local-conf-dir $WORKSPACE

sudo rm -rf $output_dir
```

```
mkdir -p $output_dir

sudo rm -rf $work_dir
mkdir -p $work_dir
cd $work_dir

sudo apt-get update -c $aptconffile

# Grab a newer version of pbuilder, because the one that ships with Lucid suffers from a bug
when using --execute
# https://bugs.launchpad.net/ubuntu/+source/pbuilder/+bug/811016
rm -f $WORKSPACE/pbuilder
wget -O $WORKSPACE/pbuilder
http://bazaar.launchpad.net/~vcs-imports/pbuilder/trunk/download/head:/pbuilder/pbuilder
chmod a+x $WORKSPACE/pbuilder

# Setup the pbuilder environment if not existing, or update
if [ ! -e $basetgz ] || [ ! -s $basetgz ]
then
  date +%s > ${TIMEDIR}/pbuilder_create_init
  #make sure the base dir exists
  sudo mkdir -p $base
  #create the base image
  sudo $WORKSPACE/pbuilder create \
       --distribution $distro \
       --aptconfdir $rootdir/etc/apt \
       --basetgz $basetgz \
       --architecture $arch \
       --mirror http://archive.ubuntu.com/ubuntu \
       --debootstrapopts "--keyring=/usr/share/keyrings/ubuntu-archive-keyring.gpg"
       date +%s > ${TIMEDIR}/pbuilder_create_end
else
  date +%s > ${TIMEDIR}/pbuilder_update_init
  sudo $WORKSPACE/pbuilder --update --basetgz $basetgz
  date +%s > ${TIMEDIR}/pbuilder_update_end
fi

# Boilerplate.
# DO NOT MODIFY
#####################################################

cat > build.sh << DELIM
date +%s > ${TIMEDIR}/build_sh_init

#####################################################
# Make project-specific changes here
#
set -ex

# get ROS repo's key, to be used both in installing prereqs here and in creating the pbuilder
chroot
apt-get install -y wget
sh -c 'echo "deb http://packages.ros.org/ros/ubuntu precise main" >
/etc/apt/sources.list.d/ros-latest.list'
wget http://packages.ros.org/ros.key -O - | apt-key add -
apt-get update

# Step 1: install everything you need
# Required stuff for Gazebo
apt-get install -y cmake build-essential debhelper libfreeimage-dev libprotoc-dev
libprotobuf-dev protobuf-compiler freeglut3-dev libcurl4-openssl-dev libtinyxml-dev libtar-dev
```

```
libtbb-dev ros-fuerte-visualization-common libxml2-dev pkg-config libqt4-dev
ros-fuerte-urdfdom libltdl-dev libboost-thread-dev libboost-signals-dev libboost-system-dev
libboost-filesystem-dev libboost-program-options-dev libboost-regex-dev libboost-iostreams-dev
cppcheck

# Step 2: configure and build
date +%s > ${TIMEDIR}/build_and_test_init

# Normal cmake routine for Gazebo
rm -rf $WORKSPACE/build $WORKSPACE/install
mkdir -p $WORKSPACE/build $WORKSPACE/install
cd $WORKSPACE/build
cmake
-DPKG_CONFIG_PATH=/opt/ros/fuerte/lib/pkgconfig:/opt/ros/fuerte/stacks/visualization_common/og
re/ogre/lib/pkgconfig -DCMAKE_INSTALL_PREFIX=$WORKSPACE/install $WORKSPACE/gazebo
make -j3
make install
. $WORKSPACE/install/share/gazebo-1.*/setup.sh
LD_LIBRARY_PATH=/opt/ros/fuerte/lib:/opt/ros/fuerte/stacks/visualization_common/ogre/ogre/lib
make test ARGS="-VV" || true

# Step 3: code check
cd $WORKSPACE/gazebo
sh tools/code_check.sh -xmldir $WORKSPACE/build/cppcheck_results || true
date +%s > ${TIMEDIR}/build_and_test_end
date +%s > ${TIMEDIR}/build_sh_end
DELIM

# Make project-specific changes here
#####################################################

sudo $WORKSPACE/pbuilder  --execute \
       --bindmounts $WORKSPACE \
       --basetgz $basetgz \
       -- build.sh

date +%s > ${TIMEDIR}/absolute_end
```

14.3 Anexo 3: código inicial de ejecución de pruebas automáticas - docker

```
#!/bin/bash -x
set -e


TIMEDIR=${WORKSPACE}/timing
mkdir -p $TIMEDIR

date +%s > ${TIMEDIR}/absolute_init

#####################################################
# Dependencies
BASE_DEPENDENCIES="build-essential \
                   cmake          \
                   debhelper      \
                   cppcheck       \
                   xsltproc       \
                   python         \
                   gnupg2         \
                   python-empy    \
                   python-argparse \
                   debhelper"

GAZEBO_BASE_DEPENDENCIES="cppcheck \
                          libfreeimage-dev      \
                          libprotoc-dev               \
                          libprotobuf-dev             \
                          protobuf-compiler           \
                          freeglut3-dev               \
                          libcurl4-openssl-dev        \
                          libtinyxml-dev              \
                          libtar-dev                  \
                          libtbb-dev                  \
                          ros-fuerte-visualization-common  \
                          ros-fuerte-urdfdom          \
                          libxml2-dev                 \
                          pkg-config                  \
                          libqt4-dev                  \
                          libltdl-dev                 \
                          libgts-dev                  \
                          libboost-thread-dev         \
                          libboost-signals-dev        \
                          libboost-system-dev         \
                          libboost-filesystem-dev     \
                          libboost-program-options-dev \
                          libboost-regex-dev          \
                          libboost-iostreams-dev      \
                          sdformat"
#####################################################

#####################################################
# Job Configuration
LINUX_DISTRO=ubuntu
DISTRO=precise
ARCH=amd64
```

```
USE_OSRF_REPO=true
USE_ROS_REPO=true
DEPENDENCY_PKGS="${BASE_DEPENDENCIES} \
               ${GAZEBO_BASE_DEPENDENCIES}"
##################################################

##################################################
# build.sh
#
cat > build.sh << DELIM
date +%s > ${TIMEDIR}/build_sh_init
# Step 2: configure and build
date +%s > ${TIMEDIR}/build_and_test_init

# Normal cmake routine for Gazebo
rm -rf $WORKSPACE/build $WORKSPACE/install
mkdir -p $WORKSPACE/build $WORKSPACE/install
cd $WORKSPACE/build
cmake
-DPKG_CONFIG_PATH=/opt/ros/fuerte/lib/pkgconfig:/opt/ros/fuerte/stacks/visualization_common/og
re/ogre/lib/pkgconfig -DCMAKE_INSTALL_PREFIX=$WORKSPACE/install $WORKSPACE/gazebo
make -j3
make install
. $WORKSPACE/install/share/gazebo-1.*/setup.sh
LD_LIBRARY_PATH=/opt/ros/fuerte/lib:/opt/ros/fuerte/stacks/visualization_common/ogre/ogre/lib
make test ARGS="-VV" || true

# Step 3: code check
cd $WORKSPACE/gazebo
sh tools/code_check.sh -xmldir $WORKSPACE/build/cppcheck_results || true
date +%s > ${TIMEDIR}/build_and_test_end
date +%s > ${TIMEDIR}/build_sh_end
DELIM
##################################################

##################################################
# Docker creation


date +%s > ${TIMEDIR}/docker_create_init

case ${LINUX_DISTRO} in
  'ubuntu')
       SOURCE_LIST_URL="http://archive.ubuntu.com/ubuntu"
       # zesty does not ship locales by default
       export DEPENDENCY_PKGS="locales ${DEPENDENCY_PKGS}"
       ;;
  *)
       echo "Unknow linux distribution: ${LINUX_DISTRO}"
       exit 1
esac

# Select the docker container depenending on the ARCH
case ${ARCH} in
  'amd64')
       FROM_VALUE=${LINUX_DISTRO}:${DISTRO}
       ;;
  'i386')
       if [[ ${LINUX_DISTRO} == 'ubuntu' ]]; then
       FROM_VALUE=osrf/${LINUX_DISTRO}_${ARCH}:${DISTRO}
       else
```

```
        FROM_VALUE=${LINUX_DISTRO}:${DISTRO}
        fi
        ;;
   'armhf' | 'arm64' )
        FROM_VALUE=osrf/${LINUX_DISTRO}_${ARCH}:${DISTRO}
        ;;
  *)
        echo "Arch unknown"
        exit 1
esac

[[ -z ${USE_OSRF_REPO} ]] && USE_OSRF_REPO=false
[[ -z ${OSRF_REPOS_TO_USE} ]] && OSRF_REPOS_TO_USE=""
[[ -z ${USE_ROS_REPO} ]] && USE_ROS_REPO=false

# depracted variable, do migration here
if [[ -z ${OSRF_REPOS_TO_USE} ]]; then
  if ${USE_OSRF_REPO}; then
        OSRF_REPOS_TO_USE="stable"
  fi
fi

DOCKER_RND_ID=$(( ( RANDOM % 10000 )  + 1 ))

cat > Dockerfile << DELIM_DOCKER
# Docker file to run build.sh

FROM ${FROM_VALUE}
MAINTAINER Jose Luis Rivero <jose.luis.rivero.partida@gmail.com>

# setup environment
ENV LANG C
ENV LC_ALL C
ENV DEBIAN_FRONTEND noninteractive
ENV DEBFULLNAME "OSRF Jenkins"
ENV DEBEMAIL "jose.luis.rivero.partida@gmail.com"
DELIM_DOCKER

# Handle special INVALIDATE_DOCKER_CACHE keyword by set a random
if [[ -n ${INVALIDATE_DOCKER_CACHE} ]]; then
cat >> Dockerfile << DELIM_DOCKER_INVALIDATE
RUN echo 'BEGIN SECTION: invalidate full docker cache'
RUN echo "Detecting content in INVALIDATE_DOCKER_CACHE. Invalidating it"
RUN echo "Invalidate cache enabled. ${DOCKER_RND_ID}"
RUN echo 'END SECTION'
DELIM_DOCKER_INVALIDATE
fi

# The redirection fails too many times using us default httpredir
if [[ ${LINUX_DISTRO} == 'debian' ]]; then
cat >> Dockerfile << DELIM_DEBIAN_APT
  RUN sed -i -e 's:httpredir:ftp.us:g' /etc/apt/sources.list
  RUN echo "deb-src http://ftp.us.debian.org/debian ${DISTRO} main" \\
                                                        >> /etc/apt/sources.list
DELIM_DEBIAN_APT
fi

if [[ ${LINUX_DISTRO} == 'ubuntu' ]]; then
  if [[ ${ARCH} != 'armhf' && ${ARCH} != 'arm64' ]]; then
cat >> Dockerfile << DELIM_DOCKER_ARCH
  # Note that main,restricted and universe are not here, only multiverse
```

```
  # main, restricted and unvierse are already setup in the original image
  RUN echo "deb ${SOURCE_LIST_URL} ${DISTRO} multiverse" \\
                                                          >> /etc/apt/sources.list && \\
        echo "deb ${SOURCE_LIST_URL} ${DISTRO}-updates main restricted universe multiverse" \\
                                                          >> /etc/apt/sources.list && \\
        echo "deb ${SOURCE_LIST_URL} ${DISTRO}-security main restricted universe multiverse" &&
\\
                                                          >> /etc/apt/sources.list
DELIM_DOCKER_ARCH
  fi
fi

# i386 image only have main by default
if [[ ${LINUX_DISTRO} == 'ubuntu' && ${ARCH} == 'i386' ]]; then
cat >> Dockerfile << DELIM_DOCKER_I386_APT
RUN echo "deb ${SOURCE_LIST_URL} ${DISTRO} restricted universe" \\
                                                          >> /etc/apt/sources.list
DELIM_DOCKER_I386_APT
fi

# Workaround for: https://bugs.launchpad.net/ubuntu/+source/systemd/+bug/1325142
if [[ ${ARCH} == 'i386' ]]; then
cat >> Dockerfile << DELIM_DOCKER_PAM_BUG
RUN echo "Workaround on i386 to bug in libpam. Needs first apt-get update"
RUN dpkg-divert --rename --add /usr/sbin/invoke-rc.d \\
       && ln -s /bin/true /usr/sbin/invoke-rc.d \\
    && apt-get update \\
       && apt-get install -y libpam-systemd \\
    && rm /usr/sbin/invoke-rc.d \\
       && dpkg-divert --rename --remove /usr/sbin/invoke-rc.d
DELIM_DOCKER_PAM_BUG
fi

# dirmngr from Yaketty on needed by apt-key
if [[ $DISTRO != 'trusty' ]] || [[ $DISTRO != 'xenial' ]]; then
cat >> Dockerfile << DELIM_DOCKER_DIRMNGR
RUN apt-get update && \\
       apt-get install -y dirmngr
DELIM_DOCKER_DIRMNGR
fi

for repo in ${OSRF_REPOS_TO_USE}; do
cat >> Dockerfile << DELIM_OSRF_REPO
RUN echo "deb http://packages.osrfoundation.org/gazebo/${LINUX_DISTRO}-${repo} ${DISTRO} main"
>\\
                                                    /etc/apt/sources.list.d/osrf.${repo}.list
RUN apt-key adv --keyserver ha.pool.sks-keyservers.net --recv-keys
D2486D2DD83DB69272AFE98867170598AF249743
DELIM_OSRF_REPO
done

if ${USE_ROS_REPO}; then
cat >> Dockerfile << DELIM_ROS_REPO
# Note that ROS uses ubuntu hardcoded in the paths of repositories
RUN echo "deb http://packages.ros.org/ros/ubuntu ${DISTRO} main" > \\
                                                    /etc/apt/sources.list.d/ros.list
RUN apt-key adv --keyserver ha.pool.sks-keyservers.net --recv-keys
421C365BD9FF1F717815A3895523BAEEB01FA116
DELIM_ROS_REPO
fi
```

```
# Packages that will be installed and cached by docker. In a non-cache
# run below, the docker script will check for the latest updates
PACKAGES_CACHE_AND_CHECK_UPDATES="${BASE_DEPENDENCIES} ${DEPENDENCY_PKGS}"

if $USE_GPU_DOCKER; then
  PACKAGES_CACHE_AND_CHECK_UPDATES="${PACKAGES_CACHE_AND_CHECK_UPDATES} ${GRAPHIC_CARD_PKG}"
fi

cat >> Dockerfile << DELIM_DOCKER3
# Invalidate cache monthly
# This is the firt big installation of packages on top of the raw image.
# The expection of updates is low and anyway it is cathed by the next
# update command below
# The rm after the fail of apt-get update is a workaround to deal with the error:
# Could not open file *_Packages.diff_Index - open (2: No such file or directory)
RUN echo "${MONTH_YEAR_STR}" \
 && (apt-get update || (rm -rf /var/lib/apt/lists/* && apt-get update)) \
 && apt-get install -y ${PACKAGES_CACHE_AND_CHECK_UPDATES} \
 && apt-get clean \
 && rm -rf /var/lib/apt/lists/*

# This is killing the cache so we get the most recent packages if there
# was any update. Note that we don't remove the apt/lists file here since
# it will make to run apt-get update again
RUN echo "Invalidating cache $(( ( RANDOM % 100000 )  + 1 ))" \
 && (apt-get update || (rm -rf /var/lib/apt/lists/* && apt-get update)) \
 && apt-get install -y ${PACKAGES_CACHE_AND_CHECK_UPDATES} \
 && apt-get clean

# Map the workspace into the container
RUN mkdir -p ${WORKSPACE}
DELIM_DOCKER3

if [[ -n ${SOFTWARE_DIR} ]]; then
cat >> Dockerfile << DELIM_DOCKER4
COPY ${SOFTWARE_DIR} ${WORKSPACE}/${SOFTWARE_DIR}
DELIM_DOCKER4
fi

cat >> Dockerfile << DELIM_WORKAROUND_91
# Workaround to issue:
# https://bitbucket.org/osrf/handsim/issue/91
RUN echo "en_GB.utf8 UTF-8" >> /etc/locale.gen
RUN locale-gen en_GB.utf8
ENV LC_ALL en_GB.utf8
ENV LANG en_GB.utf8
ENV LANGUAGE en_GB
# Docker has problems with Qt X11 MIT-SHM extension
ENV QT_X11_NO_MITSHM 1
DELIM_WORKAROUND_91

echo '# BEGIN SECTION: see build.sh script'
cat build.sh
echo '# END SECTION'

cat >> Dockerfile << DELIM_DOCKER4
COPY build.sh build.sh
RUN chmod +x build.sh
DELIM_DOCKER4
echo '# END SECTION'
```

```
echo '# BEGIN SECTION: see Dockerfile'
cat Dockerfile
echo '# END SECTION'

date +%s > ${TIMEDIR}/docker_create_end

################################################

################################################
# Docker execution

date +%s > ${TIMEDIR}/docker_execution_init
# TODO: run inside docker as a normal user and replace the sudo calls
export docker_cmd="docker"

if [[ -z $DOCKER_JOB_NAME ]]; then
        export DOCKER_JOB_NAME=${DOCKER_RND_ID}
        echo "Warning: DOCKER_JOB_NAME was not defined"
        echo " - using ${DOCKER_JOB_NAME}"
fi

# Check if the job was called from jenkins
if [[ -n ${BUILD_NUMBER} ]]; then
   export DOCKER_JOB_NAME="${DOCKER_JOB_NAME}:${BUILD_NUMBER}"
else
   # Reuse the random id
   export DOCKER_JOB_NAME="${DOCKER_JOB_NAME}:${DOCKER_RND_ID}"
fi

echo " - Using DOCKER_JOB_NAME ${DOCKER_JOB_NAME}"

export CIDFILE="${WORKSPACE}/${DOCKER_JOB_NAME}.cid"

# CIDFILE should not exists
if [[ -f ${CIDFILE} ]]; then
        echo "CIDFILE: ${CIDFILE} exists, which will make docker to fail."
        echo "Container ID file found, make sure the other container isn't running"
        exit 1
fi

export DOCKER_TAG="${DOCKER_JOB_NAME}"

# This are usually for continous integration jobs
sudo rm -fr ${WORKSPACE}/build
sudo mkdir -p ${WORKSPACE}/build

sudo docker build --tag ${DOCKER_TAG} .

# needed for docker in docker use
EXTRA_PARAMS_STR="--privileged"

# DOCKER_FIX is for workaround https://github.com/docker/docker/issues/14203
sudo ${docker_cmd} run $EXTRA_PARAMS_STR  \
               -e DOCKER_FIX=''  \
               -e WORKSPACE=${WORKSPACE} \
               -e TERM=xterm-256color \
               -v ${WORKSPACE}:${WORKSPACE} \
               -v /dev/log:/dev/log:ro \
               -v /run/log:/run/log:ro \
               --tty \
               --rm \
```

```
                ${DOCKER_TAG} \
                /bin/bash build.sh

date +%s > ${TIMEDIR}/docker_execution_end


##################################################

date +%s > ${TIMEDIR}/absolute_end
```

14.4 Anexo 4: código de detección del driver gráfico en uso

```
GRAPHIC_CARD_FOUND=false
GRAPHIC_CARD_PKG=""

export_display_variable()
{
       # Hack to found the current display (if available) two steps:
       # Check for /tmp/.X11-unix/ socket and check if the process is running
       for i in $(ls /tmp/.X11-unix/ | sed -e 's@^X@:@')
       do
       # grep can fail so let's disable the fail or error during its call
       set +e
       ps aux | grep bin/X.*$i | grep -v grep
       set -e
       if [ $? -eq 0 ] ; then
    export DISPLAY=$i
       fi
       done
}

if [ -z ${GPU_SUPPORT_NEEDED} ]; then
       GPU_SUPPORT_NEEDED=false
fi

if ! ${GPU_SUPPORT_NEEDED}; then
       return
fi

# First try to get the display variable
export_display_variable

# Check for Nvidia stuff. Using nvidia-docker no installation needed
if [ -n "$(lspci -v | grep nvidia | head -n 2 | grep "Kernel driver in use: nvidia")" ]; then
       export GRAPHIC_CARD_NAME="Nvidia"
       export GRAPHIC_CARD_FOUND=true
fi

# Check for ati stuff
if [ -n "$(lspci -v | grep "ATI" | grep "VGA")" ]; then
       # TODO search for correct version of fglrx
       export GRAPHIC_CARD_PKG=fglrx
       export GRAPHIC_CARD_NAME="ATI"
       export GRAPHIC_CARD_FOUND=true
fi

# Check for intel
if [ -n "$(lspci -v | grep "Kernel driver in use: i[0-9][0-9][0-9]")" ]; then
       export GRAPHIC_CARD_PKG="xserver-xorg-video-intel"
       export GRAPHIC_CARD_NAME="Intel"
       export GRAPHIC_CARD_FOUND=true
       # Need to run properly DRI on intel
       export EXTRA_PACKAGES="${EXTRA_PACKAGES} libgl1-mesa-dri"
fi

# Be sure that we have GPU support
if $GPU_SUPPORT_NEEDED; then
       # Check for the lack of presence of DISPLAY var
```

```
        if [[ ${DISPLAY} == "" ]]; then
        echo "GPU support needed by the script but DISPLAY var is empty"
        # Try to restart lightdm. It should stop the script in the case of failure
        sudo service lightdm restart
        # Second try to get display variable
        export_display_variable
        if [[ ${DISPLAY} == "" ]]; then
      echo "Imposible to get DISPLAY variable. Check your system"
      exit 1
        fi
        fi

        # Check if the GPU support was found when not
        if ! $GRAPHIC_CARD_FOUND; then
        echo "GPU support needed by the script but no graphic card found."
        echo "The DISPLAY variable contains: ${DISPLAY}"
        exit 1
        fi
fi
```

14.5 Anexo 5: módulo cmake detección de soporte OpenGL

Script actualmente en el código de Gazebo:
https://bitbucket.org/osrf/gazebo/src/gazebo8/cmake/CheckDRIDisplay.cmake

```
# FindDRI support
# Check for existance of glxinfo application
# Check for existance of support for pyopengl
MESSAGE(STATUS "Looking for display capabilities")

IF ((DEFINED FORCE_GRAPHIC_TESTS_COMPILATION) AND (${FORCE_GRAPHIC_TESTS_COMPILATION}))
  SET (VALID_DISPLAY TRUE)
  SET (VALID_DRI_DISPLAY TRUE)
  MESSAGE(STATUS " + Force requested. All capabilities on without checking")
  RETURN()
ENDIF()

SET (VALID_DISPLAY FALSE)
SET (VALID_DRI_DISPLAY FALSE)
SET (CHECKER_ERROR "(no glxinfo or pyopengl)")

IF((DEFINED ENV{DISPLAY}) AND NOT ("$ENV{DISPLAY}" STREQUAL ""))
  EXECUTE_PROCESS(
        COMMAND xwininfo -root
        RESULT_VARIABLE DISPLAY_FAIL_RESULT
        ERROR_QUIET
        OUTPUT_QUIET)
  IF (NOT DISPLAY_FAIL_RESULT)
        MESSAGE(STATUS " + found a display available ($DISPLAY is set)")
        SET (VALID_DISPLAY TRUE)

        # Continue check for DRI support in the display
        # Try to run glxinfo. If not found, variable will be empty
        FIND_PROGRAM(GLXINFO glxinfo)

        # If not display found, it will throw an error
        # Another grep pattern: "direct rendering:[[:space:]]*Yes[[:space:]]*"
        IF (GLXINFO)
        EXECUTE_PROCESS(
        COMMAND glxinfo
        COMMAND grep "direct rendering:[[:space:]]*Yes[[:space:]]*"
        ERROR_QUIET
        OUTPUT_VARIABLE GLX)

        IF (GLX)
        MESSAGE(STATUS " + found a valid dri display (glxinfo)")
        SET (VALID_DRI_DISPLAY TRUE)
        ELSE()
        SET (CHECKER_ERROR "using glxinfo")
        ENDIF ()
        ELSE ()
        EXECUTE_PROCESS(
        # RESULT_VARIABLE is store in a FAIL variable since the command
        # returns 0 if ok and 1 if error (inverse than cmake IF)
        COMMAND ${PROJECT_SOURCE_DIR}/tools/gl-test.py
        RESULT_VARIABLE GL_FAIL_RESULT
        ERROR_VARIABLE GL_ERROR
        OUTPUT_QUIET)
```

```
        IF (NOT GL_FAIL_RESULT)
        MESSAGE(STATUS " + found a valid dri display (pyopengl)")
        SET (VALID_DRI_DISPLAY TRUE)
        ELSE()
        # Check error string: no python module means no pyopengl
        STRING(FIND ${GL_ERROR}
                "ImportError: No module named OpenGL.GLUT" ERROR_POS)
        # -1 will imply pyopengl is present but real DRI test fails
        IF ("${ERROR_POS}" STREQUAL "-1")
        SET (CHECKER_ERROR "using pyopengl")
        ENDIF ()
        ENDIF ()
        ENDIF ()
  ENDIF ()
ENDIF ()

IF (NOT VALID_DISPLAY)
  MESSAGE(STATUS " ! valid display not found")
ENDIF ()

IF (NOT VALID_DRI_DISPLAY)
        MESSAGE(STATUS " ! valid dri display not found ${CHECKER_ERROR}")
ENDIF ()
```

14.6 Anexo 6: Jenkins DSL playground

14.7 Anexo 7: script testing para cygwin

```
#/bin/bash

echo "# BEGIN SECTION: Install script dependencies"
apt-cyg update
apt-cyg install wget mercurial tar gawk xz bzip2
echo '# END SECTION'

echo "# BEGIN SECTION: Install tool dependencies"
apt-cyg install python make cmake gdb git patch unzip pkg-config gcc-g++ libtool
echo '# END SECTION'

echo "# BEGIN SECTION: Install library dependencies"
apt-cyg install libpoco-devel libboost-devel libboost_python-devel libGLU-devel \
                libgtk2.0-devel libcurl-devel libjpeg-devel libfltk-devel \
                libX11-devel libXext-devel libfreetype-devel libxml2-devel libqhull-devel
echo '# END SECTION'

echo "# BEGIN SECTION: Prepare script sources"

# TODO: At his moment the script is located in a cygwin fixed dir
# need to check it out from the repository
CHECKOUT_PATH=${HOME}/ros_cygwin-master
SCRIPTS_INSTALL_DIR=/opt/rosscripts

# Cleanup the installs paths
mkdir -p /opt
rm -fr ${SCRIPTS_INSTALL_DIR}

# Copy script repo into /opt
cp -a ${CHECKOUT_PATH}/autobuild ${SCRIPTS_INSTALL_DIR}
echo '# END SECTION'

echo "# BEGIN SECTION: Run ros_build_isolated.bat"
cd ${SCRIPTS_INSTALL_DIR}
./build_ros_isolated.sh
echo "# END SECTION"
```

14.8 Anexo 8. config.xml generado por el prototipo de DSL

```
<!-- 1. roscygwin-ci_daily-cygwin64 -->
<project>
	<actions></actions>
	<description></description>
	<keepDependencies>false</keepDependencies>
	<properties>
	<hudson.model.ParametersDefinitionProperty>
		<parameterDefinitions>
		<hudson.model.StringParameterDefinition>
			<name>RTOOLS_BRANCH</name>
			<defaultValue>default</defaultValue>
			<description>release-tool branch to use</description>
		</hudson.model.StringParameterDefinition>
		</parameterDefinitions>
	</hudson.model.ParametersDefinitionProperty>
	</properties>
	<canRoam>false</canRoam>
	<disabled>false</disabled>
	<blockBuildWhenDownstreamBuilding>false</blockBuildWhenDownstreamBuilding>
	<blockBuildWhenUpstreamBuilding>false</blockBuildWhenUpstreamBuilding>
	<triggers>
	<hudson.triggers.TimerTrigger>
		<spec>@daily</spec>
	</hudson.triggers.TimerTrigger>
	</triggers>
	<concurrentBuild>false</concurrentBuild>
	<builders>
	<hudson.plugins.groovy.SystemGroovy>
		<bindings></bindings>
		<classpath></classpath>
		<scriptSource class='hudson.plugins.groovy.StringScriptSource'>
		<command>build.setDescription('RTOOLS_BRANCH: ' +
build.buildVariableResolver.resolve('RTOOLS_BRANCH'));</command>
		</scriptSource>
	</hudson.plugins.groovy.SystemGroovy>
	<hudson.tasks.Shell>
		<command>jenkins-scripts/cygwin/_ros1_roscygwin_compilation.bash</command>
	</hudson.tasks.Shell>
	</builders>
	<publishers>
	<hudson.plugins.textfinder.TextFinderPublisher>
		<regexp>failed</regexp>
		<alsoCheckConsoleOutput>true</alsoCheckConsoleOutput>
		<succeedIfFound>false</succeedIfFound>
		<unstableIfFound>false</unstableIfFound>
	</hudson.plugins.textfinder.TextFinderPublisher>
```

```
        </publishers>
        <buildWrappers>
        <hudson.plugins.ansicolor.AnsiColorBuildWrapper>
                <colorMapName>xterm</colorMapName>
        </hudson.plugins.ansicolor.AnsiColorBuildWrapper>
        </buildWrappers>
        <assignedNode>cygwin</assignedNode>
        <scm class='hudson.plugins.mercurial.MercurialSCM'>
        <source>http://bitbucket.org/osrf/release-tools</source>
        <modules></modules>
        <revisionType>BRANCH</revisionType>
        <revision>${RTOOLS_BRANCH}</revision>
        <clean>false</clean>
        <credentialsId></credentialsId>
        <disableChangeLog>false</disableChangeLog>
        </scm>
</project>
```

14.8 Anexo 9. Datos brutos sobre las ejecuciones de prueba entre docker y chroot Gazebo 5

14.8.1 Gazebo 5.0 chroot

Ronda 7	Update 1		
absoluto	1497871794	1497881132	155.63
build_and_test	1497872032	1497881129	151.62
build completo	1497871867	1497881129	154.37
pbuilder_update	1497871831	1497871863	0.53
Ronda 9	create 1		
absoluto	1498047736	1498056136	140.00
build_and_test	1498048426	1498056132	128.43
build completo	1498048122	1498056132	133.50
pbuilder_create	1498047867	1498048119	4.20
Ronda 10	update 2		
absoluto	1498056138	1498063479	122.35
build_and_test	1498056382	1498063476	118.23
build completo	1498056210	1498063476	121.10
pbuilder_update	1498056172	1498056206	0.57
Ronda 12	create 2		
absoluto	1499648131	1499654518	106.45
build_and_test	1499648888	1499654514	93.77
build completo	1499648567	1499654514	99.12
pbuilder_create	1499648290	1499648564	4.57

Ronda 15	create 3		
absoluto	1499738899	1499745719	113.67
build_and_test	1499739686	1499745716	100.50
build completo	1499739379	1499745716	105.62
pbuilder_create	1499739122	1499739375	4.22
Ronda 16	update 3		
absoluto	1499829952	1499837340	123.13
build_and_test	1499830242	1499837337	118.25
build completo	1499830060	1499837337	121.28
pbuilder_update	1499830024	1499830057	0.55

14.8.2 Gazebo 5.0 docker

Ronda 2	raw 1		
absoluto	1499627073	1499633235	102.70
build_and_test	1499627912	1499633229	88.62
build completo	1499627912	1499633229	88.62
docker_create	1499627073	1499627073	0.00
docker_exe	1499627073	1499633235	102.70
Ronda 3	cache 1		
absoluto	1499642131	1499648129	99.97
build_and_test	1499642871	1499648124	87.55
build completo	1499642871	1499648124	87.55
docker_create	1499642131	1499642131	0.00
docker_exe	1499642131	1499648129	99.97

Ronda 4	cache 2		
absoluto	1499654520	1499660085	92.75
build_and_test	1499654894	1499660079	86.42
build completo	1499654894	1499660079	86.42
docker_create	1499654520	1499654520	0.00
docker_exe	1499654520	1499660085	92.75
Ronda 14	raw 2		
absoluto	1499692707	1499699201	108.23
build_and_test	1499693388	1499699194	96.77
build completo	1499693388	1499699194	96.77
docker_create	1499692707	1499692707	0.00
docker_exe	1499692707	1499699201	108.23
Ronda 15	raw 3		
absoluto	1499732337	1499738897	109.33
build_and_test	1499733426	1499738890	91.07
build completo	1499733426	1499738890	91.07
docker_create	1499732337	1499732337	0.00
docker_exe	1499732337	1499738897	109.33
Ronda 16	cache 2		
absoluto	1499745721	1499751550	97.15
build_and_test	1499746100	1499751544	90.73
build completo	1499746100	1499751544	90.73
docker_create	1499745721	1499745721	0.00
docker_exe	1499745721	1499751550	97.15
Ronda 18	cache 3		

absoluto	1499823541	1499829949	106.80
build_and_test	1499823960	1499829943	99.72
build completo	1499823960	1499829943	99.72
docker_create	1499823541	1499823541	0.00
docker_exe	1499823541	1499829949	106.80